Simulation of Spin Models on Nvidia Graphics Cards using CUDA

D I P L O M A R B E I T

zur Erlangung des akademischen Grades

Diplom-Physiker
(Dipl.-Phys.)
im Fach Physik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Humboldt-Universität zu Berlin

von
Florian Wende
geboren am 06.05.1985 in Dresden

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Dr. h.c. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät I:
Prof. Dr. Andreas Herrmann

Gutachter:
1. Prof. Dr. U. Wolff
2. Prof. Dr. M. Müller-Preußker

eingereicht am: 19.08.2010
Tag der mündlichen Prüfung: 19.08.2010

Bei dem vorliegenden Dokument handelt es sich um die korrigierte Fassung meiner Diplomarbeit.

The present document is the corrected version of my diploma thesis.

Abstract

This thesis reports on simulating spin models on Nvidia graphics cards using the CUDA programming model; a particular approach for making GPGPU—General Purpose Computation on Graphics Processing Units—available for a wide range of software developers not necessarily acquainted with (massively) parallel programming.

By comparing program execution times for simulations of the Ising model and the Ising spin glass by means of the Metropolis algorithm on Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core x86 CPU—we used OpenMP to make our simulations run on all 4 execution units of the CPU—, we noticed that the Tesla C1060 performed about a factor $5-10$ faster than the Core i7-920, depending on the particular model and the accuracy of the calculations (32-bit or 64-bit).

We also investigated the reliability of GPGPU computations, especially with respect to the occurrence of soft-errors as suggested in [23]. We noticed faulty program outputs during long-time simulations of the Ising model on 'large' lattices. We were able to link these problems to overheating of the corresponding graphics cards.

Doing Monte Carlo simulations on parallel computer architectures, as was the case in this thesis, suggests to also generate random numbers in a parallel manner. We present implementations of the random number generators Ranlux and Mersenne Twister. In addition, we give an alternative and very efficient approach for producing parallel random numbers on Nvidia graphics cards. We successfully tested all random number generators used in this thesis for their quality by comparing Monte Carlo estimates against exact calculations.

Keywords:
GPGPU, CUDA, Ranlux, Mersenne Twister, Ising model, Metropolis algorithm, Ising spin glass, Swendsen-Wang cluster agorithm, single-cluster algorithm, critical exponents, soft-errors, Nvidia Tesla C1060/S1070

Zusammenfassung

Vorliegende Diplomarbeit befasst sich mit der Simulation von Spin-Modellen auf Nvidia Grafikkarten. Hierbei wird das Programmiermodell CUDA verwendet, welches einer breiten Masse von Softwareentwicklern den Zugang zu GPGPU—General Purpose Computation on Graphics Processing Units—gestattet ohne dass diese notwendigerweise mit (massiv) paralleler Programmierung vertraut sein müssen.

Im Rahmen von Vergleichen der Programmlaufzeiten für Simulationen des Ising-Modells sowie des Ising-Spin-Glases auf Nvidia Tesla C1060 Grafikkarten und einer Intel Core i7-920 x86 Quad-Core-CPU war zu vermerken, dass, abhängig vom konkreten Modell und der Rechengenauigkeit (32-bit oder 64-bit), die Tesla C1060 zwischen $5 - 10$ mal schneller arbeitete als die Core i7-920 Quad-Core-CPU.

Wir haben uns ebenfalls mit der Zuverlässigkeit von GPGPU-Rechnungen befasst, gerade in Hinblick auf das Auftreten von Soft-Errors, wie es in [23] angedeutet wird. Wir beobachteten falsche Programmausgaben bei langen Simulationen des Ising-Modells auf „großen" Gittern. Es gelang uns die auftretenden Probleme mit Überhitzung der entsprechenden Grafikkarten in Verbindung zu bringen.

Für die Durchführung von Monte-Carlo-Simulationen auf Parallelrechnerarchitekturen, wie es in vorliegender Arbeit der Fall ist, liegt es nahe auch Zufallszahlen parallel zu erzeugen. Wir präsentieren Implementierungen der Zufallszahlengeneratoren Ranlux und Mersenne Twister. Zusätzlich stellen wir eine alternative und sehr effiziente Möglichkeit vor parallele Zufallszahlen auf Nvidia Grafikkarten zu erzeugen. Alle verwendeten Zufallszahlengeneratoren wurden erfolgreich auf ihre Qualität getestet indem Monte-Carlo-Schätzer exakten Rechnungen gegenübergestellt wurden.

Schlagwörter:
GPGPU, CUDA, Ranlux, Mersenne Twister, Ising-Modell, Metropolis-Algorithmus, Ising-Spin-Glas, Swendsen-Wang Cluster-Algorithmus, Single-Cluster-Algorithmus, kritische Exponenten, Soft-Errors, Nvidia Tesla C1060/S1070

Contents

Introduction

With the increasing availability of large-capacity computers at the end of the sixties, a development started that made the computer become an essential part in many branches of today's life. Primarily driven by the increasing demand for computing power, current computers allow to process complex issues within a fraction of the time supercomputers would have required not so long ago. At present, many scientists are concerned with how to efficiently use multi-core CPUs (CPU—Central Processing Unit[1]), or even clusters made up of them, to make software benefit from the respective computing power. Here, also hardware accelerators (such as FPGAs[2] or graphics cards) have to be considered. Although the fields of application of hardware accelerators are very limited, they attract wide interest due to properties not found in standard computer hardware.

Graphics cards, for instance, provide a scalable set of data parallel execution units, combined with an extremely wide memory interface. They therefore combine supercomputer-like computing power with little space requirement and low acquisition and maintenance costs. Although this has also been the case for legacy graphics cards generations, only in the last two years fundamental changes in the graphics cards architecture allowed for an easy access of their computing power for non-graphics applications. From that point on, the use of graphics cards for general purpose computing (GPGPU—General Purpose Computation on Graphics Processing Units) became more and more popular. GPGPU is currently promoted by the two leading developers of professional graphics hardware, namely Nvidia and ATI/AMD. While both of them maintain (disjoint) programming models and software development tools, Nvidia's CUDA (Compute Unified Device Architecture) appears to be more established at present. Particularly in the field of science, several investigations on using CUDA were presented [19].

Since the application of GPGPU in scientific calculations is a rather new approach, there are questions about the gain in performance over compareable parallel CPU implementations, the programming effort to create applications for graphics cards, and also the reliability of GPGPU calculations. The goal of this diploma thesis is to study these aspects by using CUDA to perform Monte Carlo simulations of spin models on Nvidia graphics processing units (GPUs). Although similar investigations were already done [19, 1, 21], many of them attest graphics cards to absolutely outperform current multi-core CPUs, which at first sight is hard to believe, and in fact in many cases results from insufficient benchmark procedures. We thus present our own performance comparison, which allows to assess graphics card's capabilities in a more reliable way.

[1] The CPU is the portion of a computer system that carries out the instructions of a computer program. It is the primary element carrying out the computer's functions.

[2] A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence 'field-programmable'—.

After briefly introducing the subject of statistical physics (Section 1) [15, 4, 5] as well as the method of Monte Carlo simulation (Section 2), we will be concerned with CUDA (Section 3). Since CUDA is an essential part of this thesis, we will take some time to get a deeper insight into the CUDA programming model and the way how the graphics hardware processes CUDA programs [17, 18]. Before actually simulating spin models, we will be concerned with parallel random numbers (Section 4), which naturally enter the arena when doing Monte Carlo simulations on parallel computer architectures. Since deficiencies in random number sequences may significantly influence simulation results, we decided to use only well established generators. In particular, the Mersenne Twister (`dcmt`) [14] and the Ranlux random number generator [12, 13] will be ported to CUDA, and also an alternative approach for producing parallel random numbers on the GPU will be given.

All these preliminaries behind, we then consider the Ising model (Section 5), describing (anti-) ferromagnetism in statistical physics. Since the Ising model has been studied for several decades, and especially because it is exactly solveable in two dimensions and zero external magnetic field, it is well suited for gaining elementary experiences in using CUDA—we will be able to check simulation results against exact calculations or against values from literature, which will be of advantage to assure ourselves of correct working simulation programs—. We will simulate the critical Ising model by means of both the Metropolis algorithm and cluster algorithms, due to the phenomenon of critical slowing down. While using the Metropolis algorithm seems to be an obvious way to map the CUDA programming model onto the Ising spin system, porting cluster algorithms to CUDA will be a challenge. Even if the latter point could not be accomplished successfully, we will use the single-cluster algorithm on the one hand to check for subtle correlations [10] within the random number sequences of all parallel random number generators used in this thesis, and on the other hand to investigate the critical behavior of the Ising model by means of finite size scaling methods. Except for the phenomenon of critical slowing down, the latter point is not directly related to this work, but we aim to see beyond the scope of this thesis in order to get an insight into methodologies commonly used in this research field [8, 20].

Since simulating the Ising model in zero magnetic field predominantly involves integer arithmetic, assessing the floating-point performance of CUDA-capable graphics cards requires to switch to spin models that intensively involve floating-point arithmetic. A simple modification of the Ising model will lead us to the Ising spin glass (Section 6). In particular we will investigate the performance of both the GPU and the CPU in simulating the bimodal bond distributed Ising spin glass (ISG) and the Gaussian bond distributed ISG [15]. While the former one also involves pure integer arithmetic, the latter one will make the graphics card perform predominantly floating-point operations.

All in all, we will get comprehensive information on using CUDA to make simulations of spin models perform on Nvidia GPUs, and also about the gain in performance over parallelized CPU implementations. We will also see how the performance changes when switching from the Ising model to the bimodal bond distributed ISG and then to the Gaussian bond distributed ISG. Even the double-precision floating-point performance of current Nvidia GPUs will be taken into account.

Finally, we will give a short overview of our experiences with performance issues and the programming effort in order to write fast CUDA code and parallelized CPU code respectively. We will also ask the most fundamental question when considering to make use of GPGPU: How reliable is computing on GPUs? In a related section, we will mention differences between Nvidia's consumer graphics cards and the professional HPC (High-Performance Computing) Tesla graphics cards, especially with respect to applicability to scientific calculations.

1. Statistical Physics

This section recalls some basics of statistical physics. After having introduced the concept of statistical ensembles, the (anti-) ferromagnetic *Ising model* will be presented. By modifying the interactions between its constituents in such a way that both ferromagnetic and also anti-ferromagnetic interactions occur, we will arrive at the *Ising spin glass*. Both the Ising model and the Ising spin glass will be our test models for investigating GPGPU (see Section 5 and 6).

In a further section, the theory of *critical phenomena* will be motivated. In particular, this includes *real space renormalization* (detailed in Appendix B.1) as well as the *finite size scaling method*, which allows to estimate *critical exponents* (see Section 5). Although this thesis does not focus on the subject of the critical behavior of spin models, we include some basics about it in order to get an insight into methodologies commonly used in this research field.

1.1. Preliminaries

Statistical physics derives the behavior of macroscopic systems from microscopic equations of motion of the system's components. Generally this leads to an extremely large number of degrees of freedom, which in almost all cases makes it impossible to find an exact solution. Nevertheless, statistical physics is able to give predictions of the system's behavior by means of a probability apparatus.

A commonly used description of statistical systems is based on the concept of *statistical ensembles* [16, 15], which are made up of configurations $\{\mu\}$ (hereafter also referred to as states) that are compatible with the constraints the system underlies. In statistical physics it is assumed that the so-called *quasi-ergodic hypothesis*[1] is satisfied, which makes the whole time evolution of the system being given by the ensemble itself. Each of its members therefore has to appear with a weight p_μ that is equal to the probability the statistical system actually occupies the corresponding state. In thermal equilibrium, with the system being connected to a heatbath at temperature T (*canonical ensemble*), these *occupation probabilities* are given by the *Boltzmann weights*

$$p_\mu = \frac{1}{Z}\,\mathrm{e}^{-E_\mu/k_\mathrm{B}T}\,, \tag{1.1}$$

where E_μ is the total energy of state μ (given by the Hamiltonian H which describes the corresponding statistical system), k_B is Boltzmann's constant, and $Z = \sum_\mu \mathrm{e}^{-E_\mu/k_\mathrm{B}T}$

[1] The quasi-ergodic hypothesis states that over an infinitely long period of time the statistical system gets close to every state in phase space that is compatible with the constraints the system underlies.

denotes the partition function. We also define $\beta = 1/k_\mathrm{B}T$. In this framework, *equilibrium expectation values* of any quantity X then follow from

$$\langle X \rangle = \sum_\mu X_\mu p_\mu = \frac{1}{Z} \sum_\mu X_\mu \, \mathrm{e}^{-\beta E_\mu} \,, \tag{1.2}$$

where X_μ is the value of quantity X in state μ. $\langle X \rangle$ can be also written in terms of a derivative of the partition function with respect to X's canonically conjugated variable Y, provided that the Hamiltonian H contains a term $-XY$. If this is not the case, we introduce such a term, and set $Y = 0$ after the differentiation. In this way, E_μ definitely contains a term $-X_\mu Y$ which the derivative acts on [15]. $\langle X \rangle$ then can be written as

$$\begin{aligned} \langle X \rangle &= \frac{1}{Z} \sum_\mu X_\mu \, \mathrm{e}^{-\beta E_\mu} \\ &= \frac{1}{\beta Z} \frac{\partial}{\partial Y} \sum_\mu \mathrm{e}^{-\beta E_\mu} \Big|_{Y=0} = \frac{1}{\beta} \frac{\partial \ln Z}{\partial Y} \Big|_{Y=0} = -\frac{\partial F}{\partial Y} \Big|_{Y=0} \,, \end{aligned}$$

with the Gibbs free energy $F = -k_\mathrm{B}T \ln Z$.

One of the most physically interesting classes of properties is *fluctuations* in observable quantities

$$\left\langle \left(X - \langle X \rangle \right)^2 \right\rangle = \langle X^2 \rangle - \langle X \rangle^2 \,. \tag{1.3}$$

$\langle X^2 \rangle$ can be deduced from

$$\begin{aligned} \langle X^2 \rangle &= \frac{1}{\beta^2 Z} \frac{\partial^2}{\partial Y^2} \sum_\mu \mathrm{e}^{-\beta E_\mu} \Big|_{Y=0} = \frac{1}{\beta^2 Z} \frac{\partial^2 Z}{\partial Y^2} \Big|_{Y=0} \\ &= \frac{1}{\beta^2} \frac{\partial^2 \ln Z}{\partial Y^2} \Big|_{Y=0} + \left(\frac{1}{\beta} \frac{\partial \ln Z}{\partial Y} \Big|_{Y=0} \right)^2 = \frac{1}{\beta} \frac{\partial \langle X \rangle}{\partial Y} \Big|_{Y=0} + \langle X \rangle^2 \,, \end{aligned}$$

and Eq. (1.3) reads

$$\langle X^2 \rangle - \langle X \rangle^2 = \frac{1}{\beta} \frac{\partial \langle X \rangle}{\partial Y} \Big|_{Y=0} \equiv \frac{\chi}{\beta} \,. \tag{1.4}$$

Equation (1.4) is known as *linear response theorem*, with χ being the *susceptibility* of X to Y. If the statistical system exhibits a continuous phase transition, then fluctuations become very large near the transition point. In the thermodynamic limit they diverge.

1.2. The Ising model

A simple model to apply the methods of statistical physics to is the *Ising model* which describes (anti-) ferromagnetism [16]. It postulates a periodic d-dimensional lattice which is made up of magnetic dipoles, each of them associated with a spin s_i. While in real magnetic materials spins are continuously orientated, the Ising model assumes all spins to point either in one direction or in the opposite, expressed by $s_i = \{+1, -1\}$ or

equivalently $s_i = \{\text{'up'}, \text{'down'}\}$.

The Ising model is described by the following Hamiltonian

$$H = -J \sum_{\langle ij \rangle} s_i s_j - h \sum_i s_i \,, \tag{1.5}$$

where h refers to an external magnetic field, and the sum $\sum_{\langle ij \rangle}$ is over nearest neighbors. J denotes the strength of the coupling between adjacent spins ($J > 0$: ferromagnetism; $J < 0$: anti-ferromagnetism; subsequently, we consider $J > 0$)—having an interaction term being present within the Hamiltonian is an essential condition for the occurrence of a phase transition [16]—. In fact, the Ising model exhibits a phase transition, except for the one-dimensional case. In zero external magnetic field there are two phases, separated by the transition temperature T_c (hereafter referred to as critical temperature). For temperatures larger than T_c, the system is in a paramagnetic phase, whereas temperatures $T < T_c$ lead to a *spontaneous magnetization*. The latter point, and the model's simplicity as well as the fact that it is exactly solveable in one dimension, and particularly in two dimensions, makes the Ising model a standard toy model in statistical physics.

Its mean energy in thermal equilibrium is

$$\langle E \rangle = \frac{1}{Z} \sum_\mu E_\mu \, \mathrm{e}^{-\beta E_\mu} = -\frac{\partial}{\partial \beta} \ln Z \,. \tag{1.6}$$

Differentiating $\langle E \rangle$ with respect to the temperature T, keeping the system's volume fixed, gives the heat capacity C_V:

$$\begin{aligned} C_V &= \left(\frac{\partial \langle E \rangle}{\partial T} \right)_V = k_\mathrm{B} \beta^2 \left(\frac{\partial^2}{\partial \beta^2} \ln Z \right)_V = k_\mathrm{B} \beta^2 \frac{\partial}{\partial \beta} \left(-\frac{1}{Z} \sum_\mu E_\mu \, \mathrm{e}^{-\beta E_\mu} \right)_V \\ &= k_\mathrm{B} \beta^2 \left(-\frac{1}{Z^2} \Big(\sum_\mu E_\mu \, \mathrm{e}^{-\beta E_\mu} \Big)^2 + \frac{1}{Z} \sum_\mu E_\mu^2 \mathrm{e}^{-\beta E_\mu} \right) = k_\mathrm{B} \beta^2 \Big(\langle E^2 \rangle - \langle E \rangle^2 \Big) \,. \end{aligned}$$

Obviously C_V is proportional to the system's fluctuations in the energy E. With respect to what was said at the end of Section 1.1, C_V diverges near criticality. Subsequently, just to comply with literature on spin systems, we will consider the specific heat (capacity) $c_V = C_V / m$, the heat capacity per unit mass m. With $m = 1$ (hereafter our convention), c_V calculates the same as C_V, that is,

$$c_V = k_\mathrm{B} \beta^2 \big(\langle E^2 \rangle - \langle E \rangle^2 \big) \,. \tag{1.7}$$

To obtain the system's mean magnetization $\langle M \rangle$, we differentiate the Gibbs free energy F with respect to h:

$$\langle M \rangle = -\frac{\partial F}{\partial h} = \frac{1}{\beta} \frac{\partial \ln Z}{\partial h} = \frac{1}{Z} \sum_\mu \Big(\sum_i s_i \Big)_\mu \mathrm{e}^{-\beta E_\mu} = \Big\langle \sum_i s_i \Big\rangle \,. \tag{1.8}$$

In zero external magnetic field $h \equiv 0$ (subsequent explanations will be restricted to $h \equiv 0$), $\langle M \rangle$ vanishes. Except for the one-dimensional case, this can be explained as follows [15]: For $T > T_c$, there is no distinguished direction in space the spins would like to be aligned to—spin orientations rather depend on local magnetizations due to interactions between adjacent spins—. Since correlation lengths become smaller as T exceeds T_c (see Section 1.4.1), spins that are far apart behave almost independently, so that on average $s = +1$ occurs as often as $s = -1$, and hence $\langle M \rangle$ vanishes.

For temperatures $T < T_c$, spins behave collectively, which makes them point preferably up or down. As a consequence, a net magnetization appears. Since there is no external magnetic field, the system is in one of its symmetry-equivalent ground states.[2] Due to the fact that none of the two ground states is energetically favored, both of them are equiprobable, and in the limit of an infinitely long period of time the system occupies the one ground state as often as the other one (at least when the system volume is finite), and the mean magnetization vanishes.

To get information about the range the system's magnetization fluctuates around its equilibrium value, we need to consider its magnetic susceptibility χ that follows from differentiating its mean magnetization with respect to h

$$\chi = \left.\frac{\partial \langle M \rangle}{\partial h}\right|_{h=0} = \beta(\langle M^2 \rangle - \langle M \rangle^2)\,. \tag{1.9}$$

Within this subsection we were able to express system quantities in terms of expectation values. In Chapter 5 we will calculate these expectation values by means of Monte Carlo methods.

1.2.1. Exact solution of the 2D Ising model on finite squared lattices

Although there are exact solutions of both the one-dimensional Ising model and the two-dimensional Ising model in zero external magnetic field, the latter one is the more interesting of the two since it exhibits a phase transition. In more than two dimensions, exact solutions of the Ising model are still unknown. While the commonly quoted solution of the two-dimensional Ising model in zero external magnetic field, at first proposed by L. Onsager, is valid only in the thermodynamic limit, exact calculations on finite squared lattices are given by A. E. Ferdinand and M. E. Fisher [9]. According to their procedures, exact expressions of the internal energy per spin and the specific heat per spin can be obtained by evaluating the canonical partition function

$$Z_{mn} = \frac{1}{2}(2\sinh(2K))^{\frac{1}{2}mn} \sum_{i=1}^{4} Z_i(K)\,, \quad K = \frac{J}{k_\mathrm{B}T} \tag{1.10}$$

for an $(m \times n)$ squared lattice with periodic boundary conditions and coupling constants $J_x = J = J_y$ in zero external magnetic field. The partial partition functions Z_i are

[2] In zero external magnetic field, the Hamiltonian (1.5) is invariant under spin-flips $s \to -s$, that is, for each spin configuration there is another one with all spins inverted and with the same energy. Since none of them is favored, (1.5) has two ground states.

defined as follows:

$$Z_1 = \prod_{r=0}^{n-1} 2\cosh\left(\frac{1}{2}m\gamma_{2r+1}\right), \qquad Z_2 = \prod_{r=0}^{n-1} 2\sinh\left(\frac{1}{2}m\gamma_{2r+1}\right),$$
$$Z_3 = \prod_{r=0}^{n-1} 2\cosh\left(\frac{1}{2}m\gamma_{2r}\right), \qquad Z_4 = \prod_{r=0}^{n-1} 2\sinh\left(\frac{1}{2}m\gamma_{2r}\right).$$

The internal energy per spin is then given by

$$\frac{E_{mn}}{mn} = -\frac{J}{mn}\frac{\mathrm{d}\ln Z_{mn}}{\mathrm{d}K} = -J\coth(2K) - \frac{J}{mn}\left[\sum_{i=1}^{4} Z_i'\right]\left[\sum_{i=1}^{4} Z_i\right]^{-1}, \tag{1.11}$$

while the specific heat per spin is

$$\frac{(c_V)_{mn}}{k_{\mathrm{B}}mn} = \frac{K^2}{mn}\frac{\mathrm{d}^2\ln Z_{mn}}{\mathrm{d}K^2} = -2K^2\,\mathrm{csch}^2(2K) + \frac{K^2}{mn}\left[\frac{\sum_{i=1}^{4} Z_i''}{\sum_{i=1}^{4} Z_i} - \left(\frac{\sum_{i=1}^{4} Z_i'}{\sum_{i=1}^{4} Z_i}\right)^2\right], \tag{1.12}$$

where the primes denote differentiation with respect to K. Relations for the γ's and the derivatives Z_i' and Z_i'' are given in Appendix A. Equations (1.11) and (1.12) enable us to compare Monte Carlo estimates of the internal energy and the specific heat from simulating the Ising model on finite squared lattices against exact calculations.

Figure 1.1 shows the specific heat per spin as a function of β, where J and k_{B} were set to 1. As $\beta \to \beta_c = 0.5\ln(1+\sqrt{2}) = 0.44068\ldots$, c_V becomes extremely sharp. In the thermodynamic limit c_V would diverge.

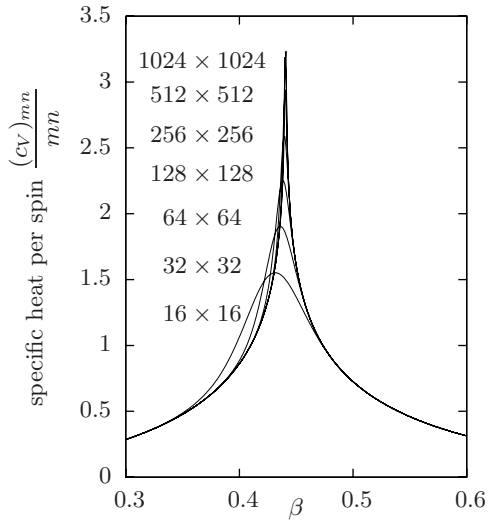

Figure 1.1.: Specific heat per spin of the two-dimensional Ising model in zero external magnetic field on finite squared lattices with extent $(m \times n)$, according to A. E. Ferdinand and M. E. Fisher [9]. J and k_{B} were set to 1.

1.3. The Ising spin glass

After this brief introduction to the (anti-) ferromagnetic Ising model, we will modify the spin-spin interactions in such a way that both ferromagnetic and anti-ferromagnetic interactions, represented by couplings J_{ij}, located on the links (*bonds*) between neighboring spins, occur. In particular, this will lead us to the Ising spin glass (ISG) [15, 11]. The widespread term 'spin glass' (amongst others encompassing amorphous substances, metallic glasses as well as glass itself) refers to a class of magnetic systems with frustrated interactions. To put it simple, *frustration* means that the product $\prod_{ij} J_{ij}$ of the links around a plaquette is negative.

As a consequence of having both ferromagnetic and anti-ferromagnetic interactions being present simultaneously, the system is unable to reach a state of lowest energy, which leads to a *large set of metastable ground states*, separated by high energy barriers the system will not be able to pass within a finite time. The latter point is the reason why, for instance, glass appears as solid material at low temperatures, and starts to melt when temperature increases.[3] The temperature that separates these two phases is usually denoted as glass temperature T_g.

Generally, one is interested in the behavior of such systems in their glassy phase, i.e. $T \leq T_g$. In order to study a glassy system on the computer, an obvious simplification is to put it onto an Ising lattice, i.e. to represent it by a lattice made up of spins pointing either in the one direction or in the opposite. To make it an Ising spin glass (ISG), the values of the couplings J_{ij} need to be both positive and negative; otherwise spins would not be frustrated and the system would not be glassy. The ISG is described by the Hamiltonian

$$H = -\sum_{\langle ij \rangle} J_{ij} s_i s_j \,, \tag{1.13}$$

which is a particular instance of the general class of models known as *Edwards-Anderson spin glasses*. Similar to the Hamiltonian of the Ising model (1.5), the sum in Eq. (1.13) is over nearest neighbors. For the *bimodal bond distributed ISG*, the couplings $J_{ij} \in \{+1, -1\}$ are randomly chosen with probability distribution

$$P(J_{ij}) = p\,\delta(J_{ij} - 1) + (1 - p)\,\delta(J_{ij} + 1)\,, \tag{1.14}$$

where p is called the *disorder parameter*. The usual ISG model corresponds to $p = 1/2$ with $[J_{ij}]$, the average over the disorder distribution, being equal to zero, i.e. $[J_{ij}] = 0$. $p \neq 1/2$ leads to $[J_{ij}] = (2p - 1) \neq 0$, and (anti-) ferromagnetic configurations are energetically favored. Alternatively, the couplings J_{ij} can be chosen to have a Gaussian distribution (*Gaussian bond distributed ISG*).

Since there are real magnetic substances that show the critical properties of the Edwards-Anderson spin glass, studying the ISG seems to be worthwile in its own right. Unfortunately, doing this on the computer is very time-consuming, especially when aiming to be comparable with literature values. In Chapter 6, we therefore decided to not

[3] Its rigidity at low temperatures follows not from a crystalline lattice, but from the energy barriers between the atoms, which therefore will not be able to move.

investigate the properties of the ISG, but to focus on appropriate implementations of the ISG using the GPU and the CPU respectively, and so to check for performance improvements by simulating glassy systems on graphics cards instead of CPUs.

1.4. An introduction to the theory of critical phenomena

As pointed out in Sections 1.1 and 1.2.1, there are quantities that show discontinuities, such as divergences, when varying the system's parameters, for instance, the temperature T. These discontinuities indicate that near some characteristic parameter configuration the system's properties will undergo deep-acting modifications. In the case of the Ising model, it is the occurrence of a spontaneous magnetization that makes the system evolve from a disordered phase ($T \gg T_c$) to an ordered phase ($T \ll T_c$).

These phase transitions are of general interest due to the universality of the critical behavior of different systems.

1.4.1. Phase transitions and critical exponents

Conventionally one divides phase transitions into those of first-order and those of higher than first-order, also called continuous phase transitions [16, 4].

First-order phase transitions involve a latent heat, which means the system absorbs or releases a fixed amount of energy. In the Ehrenfest classification of phase transitions, this results from a discontinuity in the first derivative of the Gibbs free energy with respect to a thermodynamic variable.

Continuous phase transitions involve no latent heat, but they exhibit discontinuities in higher than first-order derivatives of the Gibbs free energy. They therefore correspond to divergent susceptibilities which in turn are related to effective long-range interactions between the system's constituents.

To measure the strength of these interactions, we introduce the so called two-point spin correlation function $G^{(2)}(\boldsymbol{i},\boldsymbol{j}) = \langle \boldsymbol{s}_i \cdot \boldsymbol{s}_j \rangle$. Assuming that the lattice is isotropic, which should be valid for macroscopic systems, $G^{(2)}$ depends only on the distance $r = |\boldsymbol{i} - \boldsymbol{j}|$. Since the appearance of an external field makes $G^{(2)}$ being different from zero even if the spins $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$ do not interact, a more suitable measure for correlations due to spin-spin interactions is the *connected two-point spin correlation function*

$$G_c^{(2)}(r) = \langle \boldsymbol{s}_i \cdot \boldsymbol{s}_j \rangle - \langle \boldsymbol{s}_i \rangle \langle \boldsymbol{s}_j \rangle \,. \tag{1.15}$$

The asymptotic form of $G_c^{(2)}(r)$ for large r, compared with intermolecular distances, is given by a power law

$$G_c^{(2)}(r) \propto 1/r^{d-2+\eta} \,, \qquad r \text{ large and } T = T_c \,, \tag{1.16a}$$

where η is a critical exponent, and d is the spatial dimensionality. For $T \neq T_c$, $G_c^{(2)}(r)$ cannot be represented by a power law. In fact, for small values of the reduced tempera-

ture $t = (T - T_c)/T_c$ it behaves as

$$G_c^{(2)}(r) \propto \mathrm{e}^{-r/\xi}\,, \qquad r \text{ large and } 0 < |t| \ll 1\,, \tag{1.16b}$$

with ξ being the *correlation length*. For $T \neq T_c$, fluctuations of the system's quantities are of all sizes up to ξ, but fluctuations that are significantly larger are exceedingly rare. As $T \to T_c$, ξ grows without limits. One finds that

$$\xi \propto |t|^{-\nu}\,, \qquad T \to T_c \text{ and } h = 0\,, \tag{1.17a}$$

where ν is a further critical exponent. Actually, there are more than two of these *critical exponents*, each defined by a power law, describing the behavior of the critical system:

specific heat:	$c_V \propto \lvert t\rvert^{-\alpha}\,,$	$T \to T_c\,, h = 0\,,$	(1.17b)
magnetization:	$M \propto t^{\beta}\,,$	$T \overset{<}{\to} T_c\,, h = 0\,,$	(1.17c)
	$M \propto h^{1/\delta}\,,$	$T = T_c\,, h \to 0\,,$	(1.17d)
magnetic susceptibility:	$\chi \propto \lvert t\rvert^{-\gamma}\,,$	$T \to T_c\,, h = 0\,,$	(1.17e)
correlation function:	$G_c^{(2)}(r) \propto 1/r^{d-2+\eta}\,,$	$T = T_c\,, h = 0\,.$	(1.17f)

Knowing critical exponents of a simple-to-study system, it is their *universality* that allows to give predictions to the behavior of systems of the same universality class—systems of the same dimensionality and with the same critical exponents—without being familiar with their exact microscopic details.[4]

1.4.2. The finite size scaling method

In almost all cases it is not possible to calculate the critical exponents analytically. To get information about them, numerical methods, such as Monte Carlo methods, can be used. In so doing, the simulated systems will be finite, due to restricted capabilities of computers, and therefore they will not describe the thermodynamic limit.

While scaling laws $A \propto |t|^{-x}$ are valid only for infinitely large systems, describing the critical behavior of finite systems leads to so-called finite size scaling [5, 4]. In the following, laws of the form

$$A(t, L) \propto L^{x/\nu}\, \Psi\!\left(\frac{t}{t_0} L^{1/\nu}\right) \tag{1.18}$$

will be referred to as *finite size scaling laws*, with Ψ being a *universal scaling function* that may be different for $t > 0$ and $t < 0$. In the limit $t \to 0$, Ψ becomes a constant $\Psi(0)$, and thus is the same for all values of L. In fact, this circumstance allows to estimate

[4] This is due to the fact that reaching T_c the correlation length ξ diverges, which means that the system's behavior is dominated by effective long-range interactions. As a consequence, the exact form of short-range interactions becomes unimportant and therefore the system's behavior does not depend on the microscopic details.

critical exponents by just simulating the critical system on lattices with different extents L, and extracting $A(t=0,L)$ as a function of L.

In the limit $t \to 0$ and $L \to \infty$, the correlation length ξ of the infinite system will not be cut off by the size of the finite system any longer, and L can be replaced by the correlation length. Equation (1.18) then allows to reproduce the scaling laws of the infinite system. Substituting Eq. (1.17a) into Eq. (1.18) yields

$$A(t=0, L\to\infty) \propto L^{x/\nu} \approx \xi^{x/\nu} \propto |t|^{-x} \,.$$

Using methods of *real space renormalization*, as detailed in Appendix B.1, we can deduce finite size scaling laws from the singular part of the free energy density

$$f_s(t,h,L) = L^{-d}\,\Psi\left(\frac{t}{t_0}L^{1/\nu}, \frac{h}{h_0}L^{y_h}\right) , \tag{1.19}$$

where t_0 and h_0 are non-universal constants. The magnetic susceptibility χ, for instance, follows from:

$$\chi = \left.\frac{\partial^2 f_s}{\partial h^2}\right|_{h=0} = L^{2y_h-d}\,\Psi_\chi\!\left(\frac{t}{t_0}L^{1/\nu}\right) = L^{\gamma/\nu}\,\Psi_\chi\!\left(\frac{t}{t_0}L^{1/\nu}\right). \tag{1.20}$$

Similar scaling laws also hold for quantities other than the magnetic susceptibility. The specific heat c_V and the magnetization M, for instance, behave like $c_V = L^{\alpha/\nu}\,\Psi_{c_V}(tL^{1/\nu})$ and $M = L^{-\beta/\nu}\,\Psi_M(tL^{1/\nu})$ [15].

Corrections to finite size scaling laws are discussed in Appendix B.3. By introducing so-called irrelevant scaling variables, for instance $u_3 = u_3^0 + \mathcal{O}(t,h^2)$, the singular free energy density becomes

$$f_s(t,h,L) = L^{-d}\,\Psi\left(\frac{t}{t_0}L^{1/\nu}, \frac{h}{h_0}L^{y_h}, u_3^0 L^{-y_3}\right) ,$$

and the magnetic susceptibility is

$$\chi = L^{\gamma/\nu}\,\Psi_\chi\!\left(\frac{t}{t_0}L^{1/\nu}, u_3^0 L^{-y_3}\right) \propto L^{\gamma/\nu}\Big(1 + A_1 t L^{1/\nu} + A_2 u_3^0 L^{-y_3} + \dots\Big)\,. \tag{1.21}$$

Considering irrelevant scaling variables other than u_3 will give additional corrections to finite size scaling laws.

2. Monte Carlo methods

Applying the concepts and the techniques introduced in the previous chapter to certain models of statistical physics, such as spin models, we will be confronted with how to treat the enormous number of degrees of freedom characteristic for macroscopic many particle systems. Investigating, for instance, the properties of the three-dimensional Ising model involves an exact expression of the partition function Z, which is still unknown. Even with current computers, which only provide resources to study this model on finite lattices with, say, N lattice sites, and so render the number of degrees of freedom finite, calculating Z exactly requires to consider 2^N spin configurations, where only a very small fraction of them gives significant contributions to Z. Since the time to sum over all these configurations increases exponentially with N—this is also the case for models other than the here considered Ising model—, we require methods that allow to obtain system properties without having Z at our disposal.

In particular, we will use *Monte Carlo methods* which enable us to study the behavior of statistical systems on the computer by means of a *stochastic process*. The idea is to consecutively generate system configurations with probabilities equal to their real occupation probabilities (*importance sampling*), and so to make the system evolve through a set of states the real system would has also evolved through. Over the course of such a *Monte Carlo simulation*, measurements of system quantities need to be taken in order to estimate system properties. While the Monte Carlo approach does not explicitly require the partition function, the way system configurations are generated leads to difficulties in analyzing the measurement data (Monte Carlo data).

After introducing some basics of importance sampling and Markov processes, the *Metropolis algorithm*, as a particular choice to generate system configurations during the simulation, will be presented. Although studying the critical Ising model by means of the Metropolis algorithm would already allow to get comprehensive information about the underlying physics as well as the applicability of GPGPU to spin models, the phenomenon of *critical slowing down* will lead us to *cluster algorithms*, for certain models known to be more efficient for simulations at criticality.

At the end of this chapter, the *Gamma method*, which allows to estimate the errors of quantities deduced from Monte Carlo data, will be described.

2.1. Importance sampling and Markov processes

As already mentioned, the fact that an exact expression of the partition function Z is generally not at our disposal requires to consider methods that allow to obtain system properties without knowing Z, such as Monte Carlo methods [15]. The idea is to simulate the system dynamics, as a result of thermal fluctuations, by means of a stochastic process,

and to take measurements during the course of such a *Monte Carlo simulation*. The crux of the matter is to make the system evolve primarily through configurations with their Boltzmann weights $p_\mu = Z^{-1}\mathrm{e}^{-\beta E_\mu}$ being significantly different from zero, which is called *importance sampling*. The point is that for the majority of time the real system occupies only a very small subset of principally accessable states, namely those with large Boltzmann weights. While in the limit of an infinitely long period of time the real system occupies all accessable configurations, even those that are less probable, Monte Carlo simulations on the computer consider only a finite period of time, and hence only a finite set of configurations $\{\mu_1, \ldots, \mu_M\}$ (M does not refer to the magnetization) which then allow to describe the system's behavior only within certain limits. Since it is not possible to deduce the partition function $Z = \sum_\mu \mathrm{e}^{-\beta E_\mu}$ from $\{\mu_1, \ldots, \mu_M\}$ without causing errors due to neglecting states that were not generated during the simulation, we have to reconsider how to deduce an estimate of system quantity X from $\{X_{\mu_n}\}$. Since system configurations will be generated with probabilities p_μ instead of producing them with equal probability (simple sampling), Eq. (1.2) leads to [15]

$$X_{\mathrm{MC}} = \frac{\sum_{n=1}^{M} X_{\mu_n} p_{\mu_n}^{-1} \mathrm{e}^{-\beta E_{\mu_n}}}{\sum_{n=1}^{M} p_{\mu_n}^{-1} \mathrm{e}^{-\beta E_{\mu_n}}} = \frac{\sum_{n=1}^{M} X_{\mu_n}}{\sum_{n=1}^{M} 1} = \frac{1}{M} \sum_{n=1}^{M} X_{\mu_n} \,. \tag{2.1}$$

The *Monte Carlo estimator* X_{MC} is the mean value of the estimates $\{X_{\mu_n}\}$ of quantity X, evaluated for states $\{\mu_n\}$. The absence of the Boltzmann weights in Eq. (2.1) results from generating state μ_n with probability p_{μ_n}.

In Sections 2.2 and 2.3, we will be concerned with how to construct algorithms that sample system configurations according to their Boltzmann weights in order to produce the states $\{\mu_1, \ldots, \mu_M\}$. This directly leads to the so-called *Markov process*, a rule for randomly generating a new state from the present one, independent of the previously generated states.[1] Successive application of the Markov process then produces a so-called *Markov chain*. To treat this, we introduce probabilities $P(\mu \to \nu)$ for the system to be in state ν after previously having been in state μ—it is assumed that both state μ and state ν are compatible with the constraints the system underlies—. Since at each step the system must go somewhere, $P(\mu \to \nu)$ satisfies

$$\sum_\nu P(\mu \to \nu) = 1 \,. \tag{2.2}$$

Even it must be possible to reach any of the system's states from any other state within a finite number of Markov steps (*ergodicity assumption*). Another condition we place on the Markov process is the condition of *detailed balance* (micro-reversibility)

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu) \,. \tag{2.3}$$

Usually, the composition of Markov processes, each satisfying the condition of detailed

[1] Actually, this characterizes a first-order Markov process. There are also higher than first-order Markov processes which then consider more than one of the previously generated states in order to create a new one.

balance, only satisfies the weaker condition of *balance*

$$\sum_{\mu} p_\mu P(\mu \to \nu) = p_\nu \,, \tag{2.4}$$

or in other words: if each configuration μ appears at step n of the Markov chain according to a certain probability distribution, then it also appears according to this probability distribution at step $n+1$. Hereafter we assume $P(\mu \to \nu)$ to satisfy all these conditions.

To prove that it is the expected probability distribution that is generated by the Markov process after the system has come to equilibrium, we define the difference D_n at step n between the present distribution $W(\mu, n)$ of state μ and the expected one p_μ [4]:

$$D_n \equiv \sum_{\mu} |W(\mu, n) - p_\mu| \,.$$

If the Markov process generates states μ with probabilities p_μ, this difference should vanish in the limit $n \to \infty$:

$$\begin{aligned} D_{n+1} &= \sum_{\mu} \Big| W(\mu, n+1) - p_\mu \Big| = \sum_{\mu} \Big| \sum_{\nu} W(\nu, n) P(\nu \to \mu) - p_\mu \Big| \\ &= \sum_{\mu} \Big| \sum_{\nu} \Big(W(\nu, n) P(\nu \to \mu) - p_\nu P(\nu \to \mu) \Big) \Big| \\ &= \sum_{\mu} \sum_{\nu} \Big| \Big(W(\nu, n) - p_\nu \Big) P(\nu \to \mu) \Big| \end{aligned}$$

—using the triangle inequality—

$$\leq \sum_{\mu} \sum_{\nu} |W(\nu, n) - p_\nu| P(\nu \to \mu) = \sum_{\nu} |W(\nu, n) - p_\nu| = D_n \,.$$

Obviously, the difference D_n decreases asymptotically as n grows, i.e. $W(\mu, n) \to p_\mu$.

For later convenience we break the probabilities $P(\mu \to \nu)$ down into two parts [15]:

$$P(\mu \to \nu) = g(\mu \to \nu) A(\mu \to \nu) \,. \tag{2.5}$$

The quantity $g(\mu \to \nu)$ is the selection probability for state ν if the present state is μ. In the simplest case, given an initial state μ, $g(\mu \to \nu)$ is equal to the inverse number of states ν the system can evolve to from μ in a single Markov step. The acceptance ratio $A(\mu \to \nu)$ says that if we start off in state μ and state ν will be proposed, we would accept the state and change the system to ν a fraction of the time $A(\mu \to \nu)$.

2.2. The Metropolis algorithm for the Ising model

The Metropolis algorithm, named after N. Metropolis (1953), is a Markov chain Monte Carlo method for obtaining a sequence of random samples from a given probability distribution (here the Boltzmann probability distribution). According to the latter section,

the Metropolis algorithm has to satisfy the condition of detailed balance

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = \mathrm{e}^{-\beta(E_\nu - E_\mu)}\,, \tag{2.6}$$

and the condition of ergodicity.

Since the system primarily resides in states that have large Boltzmann weights and therefore are almost of the same energy, the Metropolis algorithm should be designed in such a way that transitions into states with energies that vary significantly from that of the other states will be suppressed. In the case of the Ising model, the most obvious way to achieve this is to propose only states that differ from the present one by just flipping a single spin [15].[2] The internal energy then changes by at most $|\Delta E| = 2zJ$, where z is the lattice coordinate number (the number of nearest neighbors). For the two-dimensional Ising model $z = 4$, and hence $|\Delta E| = 8J$.

If we consider a lattice that consists of N spins, flipping a randomly chosen spin allows to reach N different states ν. Since none of these states is favored somehow, the selection probability $g(\mu \to \nu)$ is the same for all of them

$$g(\mu \to \nu) = \frac{1}{N}\,, \tag{2.7}$$

and Eq. (2.6) becomes

$$\frac{A(\mu \to \nu)}{A(\nu \to \mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)}\,. \tag{2.8}$$

$A(\mu \to \nu) = A_0 \exp\{-\frac{1}{2}\beta(E_\nu - E_\mu)\}$, for instance, satisfies this condition. To have the algorithm as efficient as possible, the acceptance ratio $A(\mu \to \nu)$ needs to be significantly different from zero for the majority of transitions $\mu \to \nu$. Since $|E_\nu - E_\mu| \leq 2zJ$ leads to $\exp\{-\frac{1}{2}\beta(E_\nu - E_\mu)\} \leq \exp\{\beta zJ\}$, A_0 should be chosen to be equal to $\exp\{-\beta zJ\}$. Finally we arrive at

$$A(\mu \to \nu) = \mathrm{e}^{-\frac{1}{2}\beta(E_\nu - E_\mu + 2zJ)}\,. \tag{2.9}$$

Unfortunately, $A(\mu \to \nu)$ is very small for almost all transitions, as depicted in Figure (2.1). Setting up the Metropolis algorithm to be based on Eq. (2.9) would make it a very slow and inefficient algorithm since the system would spend long times in the same state. A more suitable choice is

$$A(\mu \to \nu) = \begin{cases} \mathrm{e}^{-\beta(E_\nu - E_\mu)} & \text{if } E_\nu - E_\mu > 0 \\ 1 & \text{otherwise}\,. \end{cases} \tag{2.10}$$

Obviously, detailed balance as well as the ergodicity condition are satisfied. The latter results from $A(\mu \to \nu) \neq 0$ for all transitions. It is Eq. (2.10) that defines the *Metropolis algorithm for the Ising model*. Its implementation on the computer is extremely straightforward, which is one of the main reasons for its great success. Its principal structure is

[2] This makes the Metropolis algorithm a local update algorithm, since only a single spin is altered per update step.

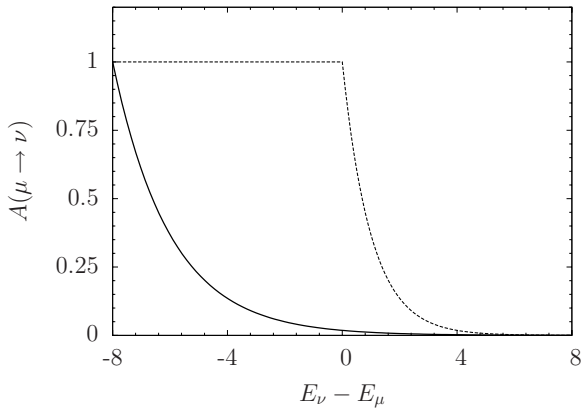

Figure 2.1.: Acceptance ratio $A(\mu \to \nu)$ according to Eq. (2.9)—straight line—and Eq. (2.10)—dashed line—for the two-dimensional Ising model. β and J are both set to 1.

given by Algorithm 1.

Metropolis algorithm for the Ising model [15]

for $k <$ iterations **do**
  choose a spin s_i at random
  consider ${s_i}^{(\nu)} = -{s_i}^{(\mu)}$ and calculate the change in energy $E_\nu - E_\mu$
  calculate $\mathcal{P} = \min[1, \mathrm{e}^{-\beta(E_\nu - E_\mu)}]$
  generate random variable r uniform on $[0, 1)$
  accept move if $r < \mathcal{P}$, i.e. set $s_i = -s_i$
end for

Algorithm 1: Structure chart of the Metropolis algorithm for the Ising model.

Due to its locality, the Metropolis algorithm can be efficiently implemented on parallel computers. Modifying Algorithm 1 in such a way that spins are not chosen at random for changing their orientation, but to alter them in a predefined order, for instance by means of a checkerboard decomposition of the lattice, we will be able to update a large set of them at the same time. How to implement the Metropolis algorithm for the Ising model by means of CUDA and OpenMP respectively, is subject of Section 5.1.

2.2.1. Critical fluctuations and critical slowing down

As already discussed in Section 1.4, the properties of the critical system are dominated by effective long-range interactions, which lead to large clusters of predominantly up- or down-pointing spins. These clusters contribute significantly to both the magnetization M and the energy E of the system, so that, as they flip from one orientation to another, they produce large fluctuations in M and E, called *critical fluctuations* [15].

The statistical errors of Monte Carlo estimates M_{MC} and E_{MC} are proportional to the size of these critical fluctuations and so are expected to grow as the size of the simulated system increases. In order to have these errors remaining the same as the lattice extent L increases, the number of measurements N has to be enlarged (the statistical error of any quantity is proportional to $N^{-1/2}$), that is, the simulation time grows.

Another point concerns the *independency of these measurements*. Since system configurations within a Markov chain each follow from the previous configuration, the number of independent measurements becomes subject to the Monte Carlo algorithm used. A quantity that provides information about the independency of measurements from consecutive system configurations is the so-called *autocorrelation function* $\Gamma_X(t)$, with t being a 'Monte Carlo time scale'. It asymptotically decreases as

$$\Gamma_X(t) \overset{t\to\infty}{\propto} \mathrm{e}^{-t/\tau_{\mathrm{exp},X}},$$

with $\tau_{\mathrm{exp},X}$ referred to as *exponential autocorrelation time* of quantity X. The number of statistically independent samples N' then is approximately

$$N' \approx \frac{N}{2\tau_{\mathrm{int},X}}, \tag{2.11}$$

where $\tau_{\mathrm{int},X}$ is the *integrated autocorrelation time* of quantity X—one has $\tau_{\mathrm{int},X} \leq \tau_{\mathrm{exp},X}$ (further information are given in Section 2.4)—. The value of the autocorrelation time near criticality strongly depends on the Monte Carlo algorithm used. The Metropolis algorithm, for instance, suffers from extremely large autocorrelation times as $T \to T_c$. This so-called *critical slowing down* makes it very inefficient to study the critical behavior of the Ising model by means of the Metropolis algorithm.

2.3. Cluster algorithms

In the case of the critical Ising model, spins are organized in clusters of size up to the lattice extent. Since spins within these clusters are surrounded by other spins pointing in the same direction, flipping them by means of the Metropolis algorithm is almost always without success, that is, almost nothing changes. To obtain independent spin configurations, each spin has to be taken into account multiple times.

Near criticality it is more efficient to make clusters of spins being the objects of interest instead of considering each spin on its own. Flipping clusters will alter the spin system much faster than flipping single spins. For the Ising model two cluster algorithms are commonly used, the single-cluster algorithm proposed by U. Wolff, and the Swendsen-Wang cluster algorithm.

2.3.1. The single-cluster algorithm for the Ising model

Since the Swendsen-Wang cluster algorithm is closely related to the single-cluster algorithm, even though the former one was first proposed, we will detail the key aspects by

Single-cluster algorithm for the Ising model [15]

choose a seed spin at random
add aligned nearest-neighbor (n.n.) spins with probability $p_{\rm add}$
while spins were added to the cluster **do**
  each added spin, adds its aligned n.n. spins with probability $p_{\rm add}$
end while
flip the cluster

Algorithm 2: Structure chart of the single-cluster algorithm for the Ising model.

considering the single-cluster algorithm for the Ising model [15].

As suggested by its name, and as can be seen from Algorithm 2, *only one cluster will be created*, with its extent depending on the probability $p_{\rm add}$ to add spins to the cluster that have the same orientation as the seed spin. In particular, $p_{\rm add}$ has to incorporate the temperature in such a way that it tends to zero for temperatures much larger than T_c, whereas it becomes equal to 1 as T becomes zero. The single-cluster algorithm also has to satisfy the condition of ergodicity and detailed balance

$$\frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)} \,. \tag{2.12}$$

To work out the acceptance ratio $A(\mu \to \nu)$, so that Eq. (2.12) is obeyed, we imagine two states of the system, say, μ and ν, as illustrated in Figure 2.2. In each of the two states there are spins outside the cluster pointing the same way as those in the cluster. Although these spins have the correct orientation they were not added to the cluster. Now suppose that for the move $\mu \to \nu$ there are m of these not-added spins. Defining the probability $(1 - p_{\rm add})$ of not adding a spin to the cluster, the probability of not adding all of these m spins, and hence the selection probability $g(\mu \to \nu)$ for the move $\mu \to \nu$, is given by $(1 - p_{\rm add})^m$. A similar expression holds for the reverse move $\nu \to \mu$, where the number of not-added spins is n. Here the reverse move, which takes us back to μ from ν, considers the same seed spin and adds the other spins to it in exactly the same way as in the forward move. Then n is equal to the number of spins around the edge of the 'forward-move-cluster' that have had the wrong orientation, and $g(\nu \to \mu) = (1 - p_{\rm add})^n$.

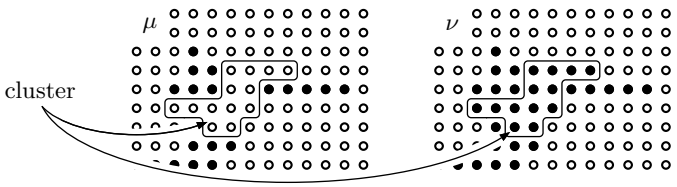

Figure 2.2.: Flipping a cluster in a simulation of the two-dimensional Ising model. The solid and open circles refer to up- and down-pointing spins in this model.

Obviously, only spins around the edge of the cluster contribute to the energy difference $E_\nu - E_\mu$, whereas the remaining spins, which are surrounded by spins with the same orientation, will give zero contribution to $E_\nu - E_\mu$. In going from $\mu \to \nu$, each of the m not-added spins contributes $+2J$ to the energy difference $E_\nu - E_\mu$, and each of the n wrong orientated spins contributes $-2J$. Thus,

$$E_\nu - E_\mu = 2J(m-n)\,.$$

Substituting these results into Eq. (2.12) then yields

$$(1-p_{\text{add}})^{m-n}\,\frac{A(\mu\to\nu)}{A(\nu\to\mu)} = \mathrm{e}^{-\beta 2J(m-n)}\,,$$

or equivalently

$$\frac{A(\mu\to\nu)}{A(\nu\to\mu)} = \left[\mathrm{e}^{2\beta J}(1-p_{\text{add}})\right]^{n-m}. \tag{2.13}$$

If we choose

$$p_{\text{add}} = 1 - \mathrm{e}^{-2\beta J}\,, \tag{2.14}$$

then the right-hand side of Eq. (2.13) is equal to 1, independent of any properties of the states μ and ν, or the temperature, or anything else at all. Since $p_{\text{add}} \in (0,1)$, every move $\mu \to \nu$ is principally possible, so that the condition of ergodicity is satisfied. Even the condition of detailed balance is obeyed. As requested above, p_{add} tends to zero for large temperatures, and becomes equal to 1 for zero temperature.

In the limit $T \to \infty$ (or $T \gg T_c$ respectively), the mean cluster size is 1, and the single-cluster algorithm becomes equivalent to the Metropolis algorithm, but it will actually be the slower of the two in this case. On the part of the single-cluster algorithm, this is due to the business of testing each of the seed spin's neighbors for possible inclusion in the cluster, whereas the Metropolis algorithm only has to decide whether to flip a single spin or not. In the low-temperature regime $T \ll T_c$, the mean cluster size tends to be equal to the number of lattice sites. Flipping the cluster makes almost all spins change their orientation. Doing this twice, the original configuration is recovered in the main. Only a small number of excitations, given by the number of rejected spins, arise. In a certain sense, this is the acting of the Metropolis algorithm for low temperatures, where its acceptance probability for a single spin-flip becomes very small. Again the Metropolis algorithm will be the faster of the two, for the same reasons as already given.

As a consequence, this leaves only the intermediate regime $T \approx T_c$ in which the single-cluster algorithm might be worth-wile. This of course is the regime the algorithm is designed to work well, and in fact it will beat the Metropolis algorithm due to much smaller autocorrelation times and thus a much better efficiency.

2.3.2. The Swendsen-Wang cluster algorithm for the Ising model

The Swendsen-Wang cluster algorithm is very similar to the single-cluster algorithm, but in contrast it *divides up the entire lattice into clusters*, for each of them using the

same probability p_{add} as for the single-cluster algorithm for possible inclusions of spins in the cluster.[3] Instead of flipping just one cluster, all of them are taken into account to be flipped, each with probability 1/2 [15].

Since the Swendsen-Wang algorithm looks for all clusters, and hence updates the whole lattice, it is more *suitable for parallel computer architectures* than the single-cluster algorithm. Nevertheless, the single-cluster algorithm is the more efficient one of the two due to a smaller dynamic exponent.[4] Another point that makes the single-cluster algorithm a priori the more efficient one of the two is the fact that the Swendsen-Wang algorithm looks for all clusters on the lattice (and thus is more resource-intensive than the single-cluster algorithm), but on average flips only half of them. This seemingly disadvantage can be partly compensated by processing the Swendsen-Wang algorithm on parallel computers. Some remarks on that can be found in Section 5.1, where also some information about porting it to CUDA will be given.

2.4. Error estimation using the Gamma method

As mentioned at the beginning of this chapter, evaluating Monte Carlo data will cause some difficulties due to *correlations between system configurations*, and thus correlations between measurement data. Applying standard methods of error analysis would suffer from ignoring these correlations, which in turn would lead to *underestimated errors of system quantities*. Since we aim to investigate the Ising model (and its critical behavior), we require methods that allow for reliably estimating system quantities and their corresponding statistical errors. For this purpose, we decided to use the *Gamma method* (Γ-method) [26], which produces more certain error estimates than commonly used binning methods.

Subsequently, we are interested in estimating the statistical error of in general non-linear functions $F = f(A_1, A_2, \ldots) = f(\{A_\alpha\})$, also referred to as 'derived quantities', with the A_α being primary observables. The idea in estimating the statistical error of f is to explicitly determine the *autocorrelation function*

$$\Gamma_{\alpha\beta}(n) = \langle (a_\alpha^i - A_\alpha)(a_\beta^{i+n} - A_\beta) \rangle \,, \quad i = 1, \ldots, M - n \,, \tag{2.15}$$

which correlates the deviations of the Monte Carlo estimates $\{a_\alpha^i\}$ of A_α with the deviations for variable β after having performed $n \geq 0$ update steps. By introducing an additional index $r = 1, \ldots, R$, which counts the number of statistically independent

[3] One possibility to construct the Swendsen-Wang clusters is to go systematically through the lattice and to start a single-cluster from each site that is not already part of an existing cluster. In this sense, dividing the entire lattice into clusters means to construct all single-clusters. This is the reason why we introduced the single-cluster algorithm first, although the focus should be on the Swendsen-Wang cluster algorithm since it is more suitable for parallel computer architectures.

[4] The dynamic exponent z, defined by $\tau \propto \xi^z$, tells us how the autocorrelation time gets longer as the correlation length diverges near the critical point. It therefore allows to compare different algorithms. A direct comparison of the dynamic exponents of the Metropolis algorithm, the single-cluster algorithm and the Swendsen-Wang cluster algorithm for the Ising model can be found in [15]. The single-cluster algorithm turns out to be more efficient than the other two.

replica,[5] Eq. (2.15) generalizes to

$$\langle (a_\alpha^{i,r} - A_\alpha)(a_\beta^{i+n,s} - A_\beta)\rangle = \delta_{rs}\Gamma_{\alpha\beta}(n)\,. \tag{2.16}$$

Estimates $\bar{\bar{a}}_\alpha$ of primary observables A_α then have to be defined in terms of per-replicum means $\bar{a}_\alpha^r = M_r^{-1}\sum_{i=1}^{M_r} a_\alpha^{i,r}$, that is,

$$\bar{\bar{a}}_\alpha = \frac{1}{M}\sum_{r=1}^{R} M_r \bar{a}_\alpha^r = \frac{1}{\sum_{r=1}^{R} M_r}\sum_{r=1}^{R}\sum_{i=1}^{M_r} a_\alpha^{i,r}\,, \tag{2.17}$$

where $M = \sum_{r=1}^{R} M_r$ is the total number of estimates. To obtain reliable estimates of the observables A_α, the estimates $\bar{\bar{a}}_\alpha$ and $\bar{a}_\alpha^r$ need to be unbiased, i.e.

$$\langle \bar{\delta}_\alpha^r\rangle = \langle \bar{a}_\alpha^r - A_\alpha\rangle = 0\,, \qquad \langle \bar{\bar{\delta}}_\alpha\rangle = \langle \bar{\bar{a}}_\alpha - A_\alpha\rangle = 0\,,$$

which is assumed hereafter. The estimates $\bar{a}_\alpha^r$ then can be described by a normal distribution, defined by the covariance matrix

$$\begin{aligned}\langle \bar{\delta}_\alpha^r \bar{\delta}_\beta^s\rangle &= \left\langle \left(\frac{1}{M_r}\sum_{i=1}^{M_r}(a_\alpha^{i,r} - A_\alpha)\right)\left(\frac{1}{M_s}\sum_{j=1}^{M_s}(a_\beta^{j,s} - A_\beta)\right)\right\rangle \\ &= \frac{1}{M_r^2}\sum_{i,j=1}^{M_r}\delta_{rs}\Gamma_{\alpha\beta}(j-i) = \frac{1}{M_r}\,\delta_{rs}C_{\alpha\beta}\times(1+\mathcal{O}(\tau/M_r))\,,\end{aligned} \tag{2.18}$$

where

$$C_{\alpha\beta} = \sum_{t=-\infty}^{+\infty}\Gamma_{\alpha\beta}(t)\,. \tag{2.19}$$

A similar expression holds for the covariance matrix of $\bar{\bar{a}}_\alpha$:

$$\langle \bar{\bar{\delta}}_\alpha \bar{\bar{\delta}}_\beta\rangle = \frac{1}{M}\,C_{\alpha\beta}\times(1+\mathcal{O}(R\tau/M))\,. \tag{2.20}$$

Equations (2.18) and (2.20) are only valid if $M_r \gg \tau$ and $M \gg \tau$. As given by Eq. (2.20), $\bar{\bar{a}}_\alpha$ differs from A_α by an error of order $1/\sqrt{M}$.

Assuming the estimates of the primary observables to be accurate enough, we define an estimator $\bar{\bar{F}} = f(\{\bar{\bar{a}}_\alpha\})$ for the derived quantity F, where the latter point should justify its Taylor expansion. Thus,

$$\bar{\bar{F}} = f(\{A_\alpha\}) + \sum_\alpha f_\alpha \bar{\bar{\delta}}_\alpha + \frac{1}{2}\sum_{\alpha,\beta} f_{\alpha\beta}\,\bar{\bar{\delta}}_\alpha\bar{\bar{\delta}}_\beta + \ldots = F + \sum_\alpha f_\alpha\bar{\bar{\delta}}_\alpha + \frac{1}{2}\sum_{\alpha,\beta} f_{\alpha\beta}\,\bar{\bar{\delta}}_\alpha\bar{\bar{\delta}}_\beta + \ldots\,,$$

[5] This refers to the number of statistically independent simulations using the same update algorithm.

with derivatives $f_\alpha = \partial f/\partial A_\alpha$, $f_{\alpha\beta} = \partial^2 f/\partial A_\alpha \partial A_\beta$ evaluated at the exact values A_1, A_2, ... The bias of this estimator is given by

$$\langle \bar{\bar{F}} - F \rangle = \sum_\alpha f_\alpha \langle \bar{\bar{\delta}}_\alpha \rangle + \frac{1}{2} \sum_{\alpha,\beta} f_{\alpha\beta} \langle \bar{\bar{\delta}}_\alpha \bar{\bar{\delta}}_\beta \rangle + \ldots \simeq \frac{1}{2M} \sum_{\alpha,\beta} f_{\alpha\beta} C_{\alpha\beta} \,. \tag{2.21}$$

The latter step uses $\langle \bar{\bar{\delta}}_\alpha \rangle \equiv 0$ as well as Eq. (2.20). In the limit $M \to \infty$, this bias tends to zero and thus is negligible.

The error σ_F is to leading order given by

$$\begin{aligned} \sigma_F^2 = \langle (\bar{\bar{F}} - F)^2 \rangle &= \left\langle \left(\sum_\alpha f_\alpha \bar{\bar{\delta}}_\alpha \right) \left(\sum_\beta f_\beta \bar{\bar{\delta}}_\beta \right) + \mathcal{O}(\bar{\bar{\delta}}^3) \right\rangle \\ &\simeq \sum_{\alpha,\beta} f_\alpha f_\beta \langle \bar{\bar{\delta}}_\alpha \bar{\bar{\delta}}_\beta \rangle \simeq \frac{1}{M} \sum_{\alpha,\beta} f_\alpha f_\beta C_{\alpha\beta} = \frac{1}{M} C_F \,. \end{aligned}$$

This can be also written as

$$\sigma_F^2 = \frac{2\tau_{\mathrm{int},F}}{M} v_F \,, \tag{2.22}$$

with the naive variance

$$v_F = \sum_{\alpha,\beta} f_\alpha f_\beta \Gamma_{\alpha\beta}(0) \,, \tag{2.23}$$

and the *integrated autocorrelation time* for F

$$\tau_{\mathrm{int},F} = \frac{1}{2v_F} \sum_{t=-\infty}^{+\infty} \sum_{\alpha,\beta} f_\alpha f_\beta \Gamma_{\alpha\beta}(t) \,. \tag{2.24}$$

With respect to Section 2.2.1, a possible interpretation of Eq. (2.22) is to understand $M/(2\tau_{\mathrm{int},F})$ (which is smaller than M if $\tau_{\mathrm{int},F} > 0.5$) as the effective number of Monte Carlo estimates. For $\tau_{\mathrm{int},F} > 0.5$, the error of quantity F then would have been underestimated if all Monte Carlo estimates would have been assumed to be independent.

The quantities $\Gamma_{\alpha\beta}(t)$, C_F and σ_F can be estimated by means of the Matlab script `UWerr.m`[6] (by U. Wolff) which evaluates the autocorrelation function of quantity F by summing over a finite t interval in order to avoid unnecessary noise from terms that vanish (only) on average. The corresponding estimator, referred to as $\bar{\bar{\Gamma}}_F(t)$, allows to define an estimator for quantity C_F

$$\bar{\bar{C}}_F(W) = \left[\bar{\bar{\Gamma}}_F(0) + 2 \sum_{t=1}^{W} \bar{\bar{\Gamma}}_F(t) \right] . \tag{2.25}$$

An appropriate window W for the summation is determined by the `UWerr` script itself, but it can be also influenced manually. Choosing W is non-trivial and should by done

[6] The `UWerr.m` script is available from `http://www.physik.hu-berlin.de/com/ALPHAsoft`. Its functionality as well as some theoretical aspects are described in [26].

with intent to minimize systematic errors from truncating the sum in Eq. (2.25). Having found an optimal W, the error σ_F then is estimated by

$$\bar{\bar{\sigma}}_F^2 = \frac{\bar{\bar{C}}_F(W)}{M}\,. \tag{2.26}$$

Amongst the mean value of quantity F and its statistical error, the script also estimates the error of the error, the autocorrelation time, and the error of the autocorrelation time. If the number of replica to be evaluated with `UWerr` is larger than 2, the script also calculates the chi-square probability

$$Q = 1 - P(\chi^2/2, (R-1)/2)$$

for the replica estimates $f(\bar{a}_\alpha^r)$ to be normally distributed. P is the incomplete Gamma function (see [22], for instance), and χ^2 is given by

$$\chi^2(K) = \sum_{r=1}^{R} \frac{(f(\bar{a}_\alpha^r) - K)^2}{C_F/M_r}\,,$$

with

$$K = \bar{F} = \frac{1}{M}\sum_{r=1}^{R} M_r f(\bar{a}_\alpha^r)$$

being a constant. Values $Q \ll 0.1$ are very unlikely to appear if the estimates $f(\bar{a}_\alpha^r)$ are normally distributed.

3. Scientific computing on Nvidia graphics cards

Currently, GPGPU (General Purpose Computation on Graphics Processing Units) is used in many research fields to perform certain computations within a fraction of the time required by standard computer hardware, such as x86 CPUs. Although both leading developers of graphics hardware, namely Nvidia and ATI/AMD, maintain (disjoint) GPGPU programming models and software development tools, Nvidia's CUDA (Compute Unified Device Architecture) appears to be more established at present. Since GPGPU on ATI/AMD graphics hardware is practically reserved for the Microsoft Windows platform,[1] we decided to use Nvidia's CUDA for software development under Linux.

In the present chapter, we first link the current graphics cards generation to already existing classes of parallel computer architectures, and then turn to the CUDA programming model. Instead of rehashing Nvidia's comprehensive CUDA programming guide [17, 18], we present the key aspects in general terms.

3.1. The GPU as highly multi-threaded many-core processor—An overview

Computer architectures can be classified by *Flynn's taxonomy* (1972). Depending on the number of concurrent instruction and data streams available in the architecture, one distinguishes between

SISD—Single Instruction, Single Data; A sequential computer which exploits no parallelism in either the instruction or data streams.

MISD—Multiple Instruction, Single Data; Heterogeneous systems operate on the same data stream and must agree on the result.

SIMD—Single Instruction, Multiple Data; A computer which exploits multiple data streams against a single instruction stream.

MIMD—Multiple Instruction, Multiple Data; Multiple autonomous processors simultaneously execute different instructions on different data.

During the past 10 years, these architectures joined up into what is commonly referred to as *heterogeneous computer architectures*. Modern supercomputers, for instance, are

[1] ATI/AMD also provides support for Linux platforms, but compared to Nvidia's longtime Linux presence it is in the early stages of development.

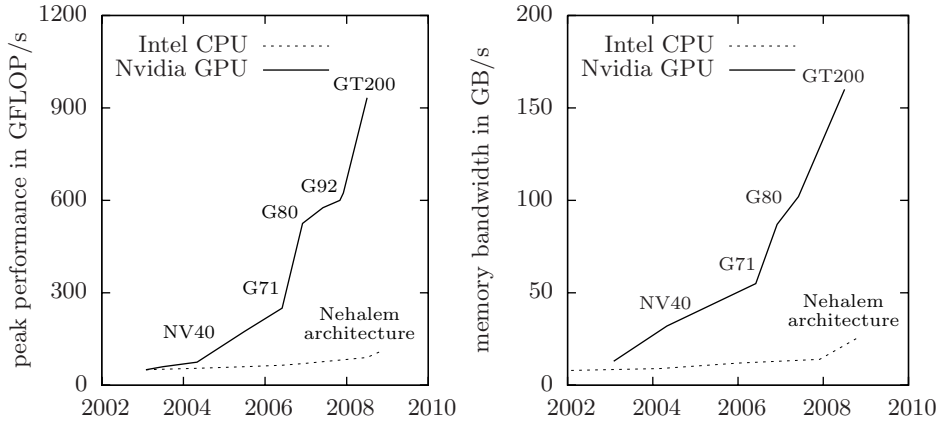

Figure 3.1.: Development of the (single-precision) peak performance and memory bandwidth of Nvidia graphics cards and Intel CPUs. The values are plotted against the years of release.

almost always made up of thousands of MIMD machines, where each of them is able to also execute 'short-vector' SIMD operations, allowing for task parallelism and data parallelism at once. However, in extensively processing large sets of data in exactly the same way, pure SIMD machines have some advantages over their heterogeneous counterparts; amongst being optimized for SIMD operations on large vectors, they also perform memory accesses in parallel.

Examples of pure SIMD machines are vector processors or array processors, such as 'APE' (Array Processor Experiment) or 'apeNEXT' at DESY Zeuthen, but also graphics processors, consisting of many data parallel execution units, originally designed to process large amounts of graphics data, are akin to SIMD. On the part of Nvidia's CUDA programming model, it is possible to have all the graphics card's execution units perform *data-parallel vector operations*, but also to have them perform independent scalar operations, which is not typical for SIMD machines. Nvidia thus refers to their graphics cards as SIMT machines—Single Instruction, Multiple Thread—, which indicates the affinity with SIMD, but also marks out thread-level parallelism[2] as an essential part of the architecture.

To get an impression of the graphics card's potential in doing massively data-parallel computations, Figure 3.1 summarizes some performance results, comparing Intel CPUs with Nvidia GPUs of the last decade. The discrepancy in the floating-point capability between CPU and GPU follows from the fact that the GPU is specialized for highly parallel computations, and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically depicted in Figure 3.2. Comparing, for instance, the Intel Core i7-920 quad-core CPU with a Nvidia Tesla C1060 graphics card, both of them are equipped with local on-chip memories—in

[2]Each thread processes its own instructions, has separate data and register states, and therefore is able to execute independently from other threads. Thread-level parallelism thus allows to hide load/store latencies if the number of threads to be executed is much larger than the number of physical execution units.

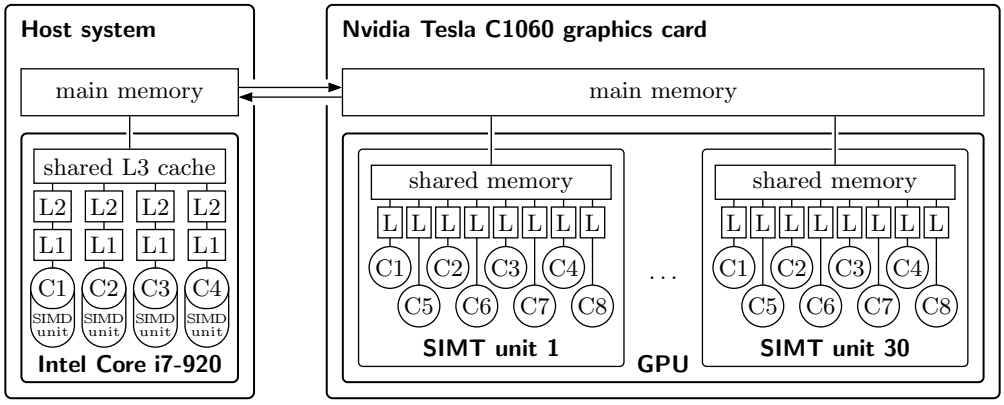

Figure 3.2.: Simplified diagram of the memory hierarchy of an Intel Core i7-920 quad-core CPU and a Nvidia Tesla C1060 graphics card. As can be seen, the Tesla C1060 provides 240 compute cores (C), but comes without any data caches, other than the Intel Core i7. Both architectures provide some kind of on-chip shared memory that allows groups of threads, mapped onto the compute cores, to cooperate among themselves and coordinate memory accesses. All compute cores have access to a uniform main memory.

Figure 3.2 denoted as L1, L2 and L—and some kind of shared memory, but only the CPU provides a real cache hierarchy. All compute cores of both the CPU and the GPU have access to respective uniform main memories, which of course are physically not the same.

To compensate for non-cached memory requests, and so to achieve high data transfer rates and high compute performance, the GPU requires the ratio of arithmetic operations per memory operation to be significantly larger than one. The idea is to hide memory access latencies with computations instead of loading data from main memory into data caches. From the programmers point of view, a graphics card program therefore needs to run a total number of threads which exceeds the number of physical execution units available on the graphics card (see Section 3.2). Applications that may take advantage of this approach thus should allow to transparently partition the problem into fine-grained sub-problems that can be solved independently by a large set of threads which are concurrently processed on the graphics card's physical execution units—the Tesla C1060 graphics card provides 240 physical execution units—.

Since the GPU is able to schedule an extremely large number of threads with zero overhead, the programmer is free to scale the problem to any number of sub-problems, independent of the number of physical execution units. In turn, the programmer has to make sure that all of these threads request data from main memory with appropriate memory access patterns, and memory accesses do not result in *race conditions*,[3] which become likely the more threads execute shared data in parallel. In Chapter 5, we will be concerned with simulating the Ising model by means of the checkerboard procedure, which consecutively applies the Metropolis algorithm to disjoint subsets of spins in order

[3] A race condition or race hazard is a flaw in an electronic system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.

to update them in parallel without causing race conditions during the updates.

To actually have graphics card programs perform well on the graphics hardware, the programmer also has to manage the GPU's on-chip shared memory, which on the part of the Intel Core i7-920 (here it is the shared L3 data cache) is scheduled by the CPU itself. On the one hand, it is of advantage to have the possibility to directly load/write data from/into this extremely fast memory from within CUDA programs, but on the other hand appropriate programs need to be optimized for using the shared memory, which again means to choose appropriate memory access patterns and prevent race conditions.

Writing graphics cards programs therefore becomes really different from writing parallel programs for the CPU, although fundamental programming aspects are similar. In the following section, we will be concerned with the CUDA programming model, which at its core provides functionalities already known from established parallel programming APIs such as OpenMP, Pthreads (POSIX Threads) or MPI.

3.2. The CUDA programming model

On the part of Nvidia it is the Tesla (SIMT) architecture (in particular the introduction of the 'unified shader'[4] in 2006) that allows to use graphics cards for general purpose computing rather than for 3D rendering. Thereby the CUDA API (API—Application Programming Interface) acts as an interface that makes adjusted `C/C++` programs run on CUDA-capable Nvidia graphics cards. At its core are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronizations. In a certain sense, this is reminiscent of the OpenMP API or the Pthreads API, which in fact provide similar abstractions.

CUDA extends the `C` programming language by allowing the programmer to define `C` functions, hereafter referred to as *kernels*, that, when called, are executed N times in parallel on the graphics card by N different CUDA threads, as opposed to only once like regular `C` functions. A CUDA thread therefore corresponds to a copy of the kernel, which in terms of *thread-level parallelism* executes independently from other threads, with its own instruction operands, and data and register states, but in most cases it performs the same instructions (data manipulations) as all the other CUDA threads—this is meant by SIMT—.

CUDA assumes all CUDA threads to execute on a physically separate device, here the graphics card, that operates as a coprocessor to the host running the `C` program. Calling a kernel within this program makes the host start a subprogram that is executed on the graphics card while the host immediately continues the `C` program until further kernel invocations, that is, CUDA kernels are non-blocking, as depicted in Figure 3.3.

A kernel is defined using the `__global__` *function type qualifier*, and the number of CUDA threads that execute the kernel is specified using the `<<<...,...>>>` syntax:

[4] In the field of computer graphics, a shader is a set of (binary) instructions used primarily to calculate rendering effects on graphics hardware. While early shader models used very different instruction sets for different types of shaders, later shader models reduced the differences, approaching the unified shader model.

```
// kernel definition
__global__ void addMat(float A[N][N], float B[N][N], float C[N][N]) {
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;
  if((i<N)&&(j<N)) C[i][j]=A[i][j]+B[i][j];
}

// main function
int main() {
  ...
  dim3 block(16,16);
  dim3 grid((N+block.x-1)/block.x,(N+block.y-1)/block.y);
  // kernel invocation
  addMat<<<grid,block>>>(A,B,C);
  ...
}
```

In this short (and rather puristic) example, each CUDA thread that executes the kernel `addMat()` is mapped onto unique data objects (`..[i][j]`) using the built-in `threadIdx`, `blockIdx`, and `blockDim` variables. Each thread then performs one pair-wise addition. The number of threads that execute the kernel `addMat()` is specified by means of the

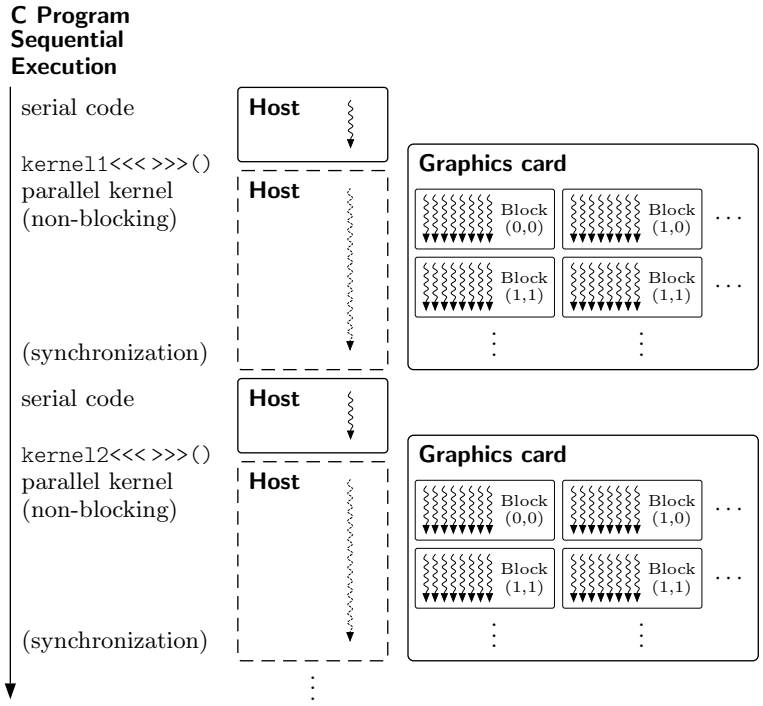

Figure 3.3.: Execution of a CUDA program. Serial code executes on the host while parallel code executes on the graphics card. Wavy arrows denote threads.

<<<. . . , . . .>>> syntax. The idea behind this new syntax is to organize all CUDA threads in groups, so-called *thread blocks*, which in turn form an up to two-dimensional *grid* of thread blocks. Due to technical reasons, thread blocks are restricted to contain at most 512 threads—therefore the necessity of a grid of thread blocks if, for instance, more than 512 CUDA threads are required—. As mentioned, CUDA threads within a thread block can be identified using a one-dimensional, two-dimensional or three-dimensional index, accessable through the built-in `threadIdx` variable. Similarly, each thread block within the grid can be identified by a one-dimensional or two-dimensional index, accessible within the kernel through the built-in `blockIdx` variable. Relating CUDA threads to data objects by means of a unique thread ID, as a function of these built-in variables, the grid of up to three-dimensional thread blocks provides a natural way to perform computations across the elements of a vector, a matrix, or a field.

During their execution CUDA threads may access data from multiple memory spaces, as schematically illustrated in Figure 3.2. All threads have a private local memory (L) and they have access to the same (non-cached) *global memory*. Each thread block has a *shared memory* visible to all threads of the block and with the same lifetime as the block. In addition all threads have access to the same *constant memory* and the same *texture memory*, which are (cached) read-only memories, optimized for different memory usages (they are not depicted in Figure 3.2).

Threads within a block can cooperate among themselves by sharing data through the shared memory and synchronizing their execution to coordinate memory accesses. When having specified *synchronization points* in the kernel by calling the `__syncthreads()` intrinsic function, all threads in the block must wait before any are allowed to proceed. Unfortunately, it is not provided to synchronize threads within different thread blocks. Only the termination of the kernel ensures that all threads are done.

Since kernels can be executed by multiple equally-shaped thread blocks, it is required that thread blocks execute independently, that is, it must be possible to execute them in any order, in parallel or in series—SIMD machines vary on this point in such a way that they execute data elements in a predefined order—. It therefore becomes possible to schedule thread blocks in any order across any number of processor cores, enabling programmers to write scalable code. On the other hand, it requires the programmer to exercise great care in writing CUDA programs, due to the occurrence of race conditions which may cause irreproducible program output. The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of execution units in the system, which it can greatly exceed.

Code-examples that describe how to use CUDA for simulations of, for instance, spin models are given in the appendixes. These codes can be used to become acquainted with CUDA. For reasons of clarity, we do not list source codes within subsequent chapters.

3.3. Nvidia Tesla architecture

Almost all CUDA programs map data parallel computational tasks onto a customized grid of thread blocks making the corresponding CUDA threads perform a set of opera-

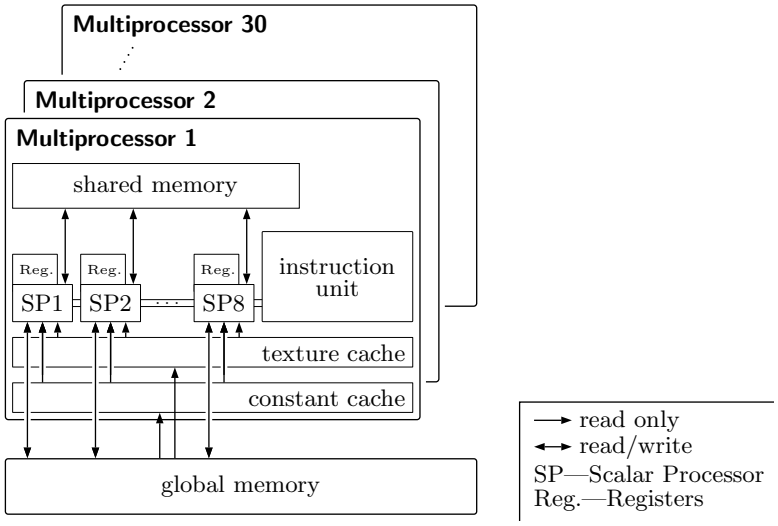

Figure 3.4.: Illustration of the multiprocessors Nvidia's Tesla architecture is based on. Each multiprocessor consists of 8 scalar processors (execution units), two special function units for transcendentals, a multithreaded instruction unit (thread scheduler), a double-precision unit, and on-chip shared memory. There is also a 'constant' cache and a 'texture cache'.

tions on appropriate data. To have these operations being executed as fast as possible, we need to know how the graphics hardware processes CUDA threads. We therefore consider Nvidia's *Tesla architecture* built around a scalable array of multithreaded *streaming multiprocessors*. Each of these multiprocessors consists of 8 *scalar processors* (execution units), two special function units for transcendentals, a multithreaded instruction unit (thread scheduler), and on-chip shared memory (see Figure 3.4). Graphics cards based on the current Tesla architecture, such as Nvidia Tesla C1060 or Nvidia GeForce GTX280/285, provide 30 of these multiprocessors, each equipped with one unit for computations with double-precision.

When a CUDA program on the host CPU invokes a kernel, the blocks of the corresponding grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. Depending on the geometry of the thread blocks, each multiprocessor is able to schedule up to 1024 CUDA threads in a concurrent manner, distributed over up to 8 thread blocks. Step-by-step, these threads, grouped into so-called *warps* (made up of 32 threads), are mapped onto the multiprocessor's 8 scalar processors, which execute them in parallel.[5] To minimize idle cycles of its scalar processors, the multiprocessors's thread scheduler continually switches between different warps in order to always have threads that are

[5]If threads within the same warp diverge in their execution paths, for instance due to datadependent conditional branches, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp. Different warps execute independently regardless of whether they are executing common or disjointed code paths.

ready to be executed while other threads run idle.

In a certain sense, this way of scheduling CUDA threads is necessary to compensate for memory requests from the non-cached global memory, regardless of whether graphics cards provide (theoretically) a very high memory bandwidth. If the number of threads exceeds the number of scalar processors, idle cycles due to memory request become negligible and the graphics card performs at its best. By implication, CUDA programs that use a number of threads that is only of the same order as the number of available scalar processors, will not efficiently use the compute capacity of the graphics card.

Performance issues are detailed in Appendix C. Basically they concern aspects of how to efficiently request data from different memory layers of the graphics card as well as how to design CUDA programs for high instruction troughput in order to assure fast thread execution.

Table 3.1 lists hardware specifications of Nvidia Tesla C1060 graphics cards and Nvidia GeForce GTX285 consumer graphics cards. All information are from the Nvidia homepage (`http://www.nvidia.com`) and the PNY homepage (`http://www3.pny.com/`).

	Tesla C1060	GeForce GTX285
Graphics chip	GT200(b)	GT200b
Scalar processors	240	240
GPU clock frequency	602MHz	648MHz
Shader clock frequency	1300MHz	1476MHz
Memory clock frequency	800MHz	1242MHz
Amount of main memory	4GB	1GB
Memory interface	512-bit	512-bit
Memory bandwidth	102GB/s	159GB/s
Peak performance (single precision)	933 GFLOP/s	1059 GFLOP/s
Peak performance (double-precision)	78 GFLOP/s	89 GFLOP/s
IEEE754 conform	yes, but not 100%	yes, but not 100%
Power consumption	< 200W, 160W normal	204W
Price (June, 2010)	$\approx$ 1200 Euro	$\approx$ 280 Euro

Table 3.1.: Hardware specifications of Nvidia Tesla C1060 and Nvidia GeForce GTX285 graphics cards.

3.4. Benchmarking

Throughout this diploma thesis, a cluster of 16 Nvidia Tesla C1060 graphics cards, and a workstation equipped with 2 Nvidia GeForce GTX285 graphics cards were used for GPU simulations, where both GeForce GTX285 graphics cards were underclocked. In particular, their default clock frequencies were adjusted to be the same as those of Tesla C1060 graphics cards. Reasons why this became necessary are given in Chapter 7.

On the part of the host system, the current Intel Nehalem processor generation was used for CPU simulations. In particular, we used a cluster consisting of 8 workstations, each equipped with 2 Intel Xeon E5520 (@2.27GHz) quad-core processors and 24GB main memory, and a workstation equipped with an Intel Core i7-920 (@2.67GHz) quad-core processor and 12GB main memory. All workstations ran a 64-bit Debian Linux.

Machine/Hardware	Owner/Institution
16 × Nvidia Tesla C1060	SGI—Silicon Graphics Inc.[6] Chippewa Falls, Wisconsin, USA
16 × Intel Xeon E5520	Humboldt Universität zu Berlin Department of Physics
2 × Nvidia GTX285	Florian Wende (privately owned)
1 × Intel Core i7-920	Florian Wende (privately owned)

To obtain serious benchmark results, all CPU programs were optimized by means of OpenMP and SSE[7] (by means of inline assembly, SSE-intrinsics, or even by rearranging compute intensive code segments to allow the compiler to optimize for SSE), that is, we had at least 4 threads to fully load all available CPU cores. Since we aim to get reliable information about the capabilities of current x86 quad-core CPUs[8] and GPUs, we deem it insufficient to run less optimized single-core programs on legacy quad-core CPUs, and then to compare program execution times with those of highly optimized GPU programs running on up-to-date high-end graphics cards, as done by many authors of 'extremely-high-speedups-over-current-quad-core-CPUs' papers [7, 21]. From our point of view, a fair comparison between CPU and GPU requires to not only spend a plenty of time in optimizing the code for the GPU, but also to do the same on the part of the code for the CPU.

All CPU programs throughout this thesis were compiled with the Intel compiler `icpc` (version 11.1) using optimization level `-fast`[9] and compiler flags `-axSSE4.2`, `-openmp`, `-static-intel`. CUDA codes were compiled with Nvidia's `nvcc` compiler wrapper.

[6] At this point, we want to give thanks to Dr. T. Steinke from the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) for providing us with an account at the SGI Chippewa Falls GPU-cluster. Further thanks go to SGI, even though we never had intensive contact to them.

[7] SSE (Streaming SIMD Extensions) is a SIMD instruction set extension to the x86 architecture, designed by Intel. It allows to process 4 32-bit words at the same time.

[8] We assume AMD x86 quad-core CPUs to give benchmark results that are comparable to those taken with equivalent Intel x86 quad-core CPUs.

[9] Before having finally switched to optimization level `-fast`, all codes were checked to give exactly the same output as having been compiled with much lower optimization levels.

4. Parallel random number generation

Random numbers frequently appear during the course of Monte Carlo simulations. Since generating them requires a non-negligible amount of time, it is very important for practical simulations to be able to generate them as fast as possible.

Throughout this chapter, the focus will be on the parallel generation of random numbers.[1] Since this is a non-trivial issue—subtle correlations between different random number sequences, for instance, may lead to deficiencies in the simulation results—only well established generators will be used. In particular, this is the Ranlux random number generator, proposed by M. Lüscher [12], and the Mersenne Twister random number generator (using `dcmt`), proposed by M. Matsumoto and T. Nishimura [14]. After briefly introducing the corresponding algorithms as well as giving some remarks on their implementation by means of CUDA and OpenMP respectively, an own approach in generating parallel random numbers using CUDA will be described.

4.1. Ranlux

The Ranlux random number generator, proposed by M. Lüscher in 1993, is based on the *subtract-with-borrow random number generator* by Marsaglia and Zaman (1991). It uses the recursion

$$y_i = (x_{i-s} - x_{i-r} - c_{i-1})\,, \qquad \begin{array}{lll} x_i = y_i\,, & c_i = 0 & \text{if } y_i \geq 0\,, \\ x_i = y_i + m\,, & c_i = 1 & \text{if } y_i < 0\,. \end{array} \tag{4.1}$$

To generate a new random number, this algorithm uses random numbers that have been produced s and r cycles ago. Choosing adequate values for s, r and m, astronomical periods can be achieved for the random number sequences generated with the subtract-with-borrow random number generator. Unfortunately, this generator suffers from rather poor statistical properties due to short-range correlations between random numbers within the same sequence. Lüscher proposed to produce a certain number of random samples and to use only a fraction of them in order to obtain better statistical properties. An implementation of the corresponding generator was first developed by F. James, who introduced the *luxury level* parameter for the fraction r/p of random numbers that are actually used. He also introduced the name *Ranlux* for this generator, characterized by the parameters $s = 10$, $r = 24$ and $m = 2^{24}$. For certain values $p \gtrapprox 200$, Lüscher attests Ranlux to generate high-quality random number sequences [12].

[1] Although it is impossible to make the computer produce real random numbers by means of deterministic algorithms, the output of so-called pseudo random number generators looks random. When using the term 'random number' we actually mean 'pseudo random number' .

Although random number generation with Ranlux becomes very time-consuming with increasing luxury level, its high-quality random numbers justify its use in several fields of application. One of its merits is to use it for parallel random number generation by just running several Ranlux instances with different seed values. The generated random number sequences are mathematically guaranteed to be uncorrelated.

Some remarks on the implementation

Implementing Ranlux on the CPU is straightforward since a highly optimized implementation using the `C` programming language is already available [12, 13]. To make this implementation provide the possibility to run more than just one Ranlux instance at the same time (within the same program), we adapted it to the `C++` programming language. In particular, we defined a `C++` class, including methods that implement the functionality of the current Ranlux implementation (version 3.3). This class then allows to instantiate several Ranlux objects that can be processed in parallel, for instance, by means of OpenMP.

Porting Ranlux to CUDA requires to reconsider SSE optimizations that are used in the current `C` implementation. Since Nvidia graphics cards do not support SSE, we mimicked this feature by using 4 CUDA threads to process Ranlux's random states in an SSE-like fashion. Each CUDA thread therefore runs its own sub-Ranlux instance.

To fully load the GPU, the number of sub-Ranlux instances per thread block should be larger than the number of scalar processors per multiprocessor (this is 8), and the number of thread blocks should be larger than the number of multiprocessors (30 on Nvidia Tesla C1060 and GeForce GTX280/285 respectively). Since CUDA threads can use a maximum of 16kB low latency shared memory per multiprocessor, frequently used data, such as the 24 random states (96Byte) per sub-Ranlux instance, is copied into the shared memory to avoid expensive memory calls while generating random numbers. To have each multiprocessor schedule a maximum number of 8 thread blocks at the same time, the number of threads per thread block calculates as $16384\text{Byte}/(8 \times 96\text{Byte}) = 21.\bar{3} \rightarrow 16$. During the course of many simulations, we found that exactly this configuration yields best performance. On Nvidia Tesla C1060 graphics cards this is a total of $30 \times 8 = 240$ thread blocks, each containing 16 CUDA threads. All in all, there are 4×960 sub-Ranlux instances, generating exactly the same random number sequences as the equivalent 960 SSE optimized CPU Ranlux instances.

The source code of our Ranlux implementation using CUDA is given by Listing D.1 in Appendix D.1.

Execution times

To estimate the mean execution time to generate a single random number, both the CPU implementations and the GPU implementations were to generate 960×10^5 random numbers, that is, 10^5 samples per Ranlux instance.[2] To obtain halfway reliable execution

[2]The CPU implementation also ran 960 Ranlux instances in order to produce exactly the same random number sequences as the GPU implementation. Producing random numbers this way, also ensures that

times, all measurements were repeated 10 times.

Table 4.1 lists the mean execution times in nanoseconds per random number $\bar{t}_{\text{sample}}$ as well as the number of samples per second $\bar{r}_{\text{sample}}$.

		Nvidia Tesla C1060		Intel Core i7-920	
luxury Level	precision	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$
0 ($p = 109$)	single	0.80	1.25	2.12	0.47
	double	—	—	—	—
1 ($p = 202$)	single	1.39	0.72	3.07	0.33
	double	3.35	0.30	5.24	0.19
2 ($p = 397$)	single	2.62	0.38	5.07	0.20
	double	5.83	0.17	9.81	0.10

Table 4.1.: Mean execution times in nanoseconds per random number $\bar{t}_{\text{sample}}$ as well as the number of samples per second $\bar{r}_{\text{sample}}$. Both the GPU and the CPU each ran 960 Ranlux instances concurrently. All measurements were repeated 10 times.

The Tesla C1060 graphics card is able to produce the same random number sequences about $2-3$ times faster than an Intel Core i7-920. In contrast to our benchmark results, speedups larger than ours by more than a factor 65 have been presented in [7]. While the author's implementation of the Ranlux random number generator for the GPU[3] gives execution times compareable to ours, his Ranlux implementation for the CPU is not multithreaded, and it gives very poor execution times even on a single CPU core. The author states a mean execution time per random number of about 200ns when using Ranlux with luxury level 1 ($p = 202$) on an Intel Core 2 Quad Q6600 quad-core CPU. We suppose that requesting for single random numbers, as done by the author, is inappropriate for fast random number generation on the CPU.

In a certain sense, [7] is a good example of how not to compare the performance of the GPU with that of the CPU. As mentioned in Section 3.4, it is absolutely insufficient to run less optimized single-core programs on legacy multi-core CPUs, and then to compare program execution times with those of highly optimized GPU programs running on up-to-date high-end graphics cards. In so doing, speedups are systematically too large by orders of magnitude.

4.2. Mersenne Twister

The Mersenne Twister random number generator, proposed by M. Matsumoto and T. Nishimura [14], provides for fast generation of very high-quality random numbers. It is

the CPU's SSE engines are fully loaded over a long period of time, and thus give optimal performance.

[3]The author uses ATI/AMD consumer graphics cards, which makes it more difficult to compare his results with ours.

based on the recursion

$$\boldsymbol{x}_{k+n} = \boldsymbol{x}_{k+m} \oplus (\boldsymbol{x}_k^u | \boldsymbol{x}_{k+1}^l) A\,, \quad k = 0, 1, 2, \ldots\,, \quad 1 \leq m < n\,, \tag{4.2}$$

where $\boldsymbol{x}_k$ is a word vector over the two-element field $\mathbb{F}_2 = \{0, 1\}$, identified with machine words of size w. $\boldsymbol{x}_k^u | \boldsymbol{x}_{k+1}^l$ is the concatenation of the upper $w - r$ bits of $\boldsymbol{x}_k$ and the lower r bits of $\boldsymbol{x}_{k+1}$, that is,

$$\boldsymbol{x}_k^u | \boldsymbol{x}_{k+1}^l = (x_k^{w-1}, \ldots, x_k^r, x_{k+1}^{r-1}, \ldots, x_{k+1}^0)\,. \tag{4.3}$$

Then this vector is multiplied by matrix A from the right. The new word vector $\boldsymbol{x}_{k+n}$ results from just adding $\boldsymbol{x}_{k+m}$ bitwise ($\oplus$). Starting with bit vectors $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}$ the recursion (4.2) then allows to successively generate new bit vectors. To assure the good statistical properties of the Mersenne Twister algorithm, each generated word is multiplied by the so-called *tempering matrix* T from the right, that is, $\boldsymbol{x} \mapsto \boldsymbol{z} = \boldsymbol{x}T$. This tempering process can be realized by the following successive transformations:

$$\begin{aligned} \boldsymbol{y} &:= \boldsymbol{x} \oplus (\boldsymbol{x} \gg u) \\ \boldsymbol{y} &:= \boldsymbol{y} \oplus ((\boldsymbol{y} \ll s) \text{ AND } \boldsymbol{b}) \\ \boldsymbol{y} &:= \boldsymbol{y} \oplus ((\boldsymbol{y} \ll t) \text{ AND } \boldsymbol{c}) \\ \boldsymbol{z} &:= \boldsymbol{y} \oplus (\boldsymbol{x} \gg l)\,, \end{aligned} \tag{4.4}$$

where l, s, t and u are integers, $\boldsymbol{b}$ and $\boldsymbol{c}$ are suitable bitmasks of word size, and $(\boldsymbol{x} \gg u)$ denotes the u-bit shiftright ($(\boldsymbol{x} \ll u)$ denotes the u-bit shiftleft). An implementation of the Mersenne Twister algorithm is available from the authors.

To use several Mersenne Twister instances in parallel, a special program, called `dcmt`, needs to be used. `dcmt` encodes the IDs of a certain number of Mersenne Twisters as well as the aimed period of their random number sequences into unique sets of parameters that define the computation of $(\boldsymbol{x}_k^u | \boldsymbol{x}_{k+1}^l) A$ and $\boldsymbol{x} \mapsto \boldsymbol{z} = \boldsymbol{x}T$. Random number sequences from different Mersenne Twister instances, set up with `dcmt`, then are guaranteed to be independent from one another.

Our implementations of the Mersenne Twister algorithm using `dcmt` were written to be absolutely conform to the implementation by Matsumoto and Nishimura. We checked for correctness of our implementations.

Although Matsumoto and Nishimura do not provide a double-precision version of `dcmt`, it is possible to combine two single-precision random numbers to obtain one double-precision sample. Our implementation using CUDA combines two single-precision random numbers generated by the same Mersenne Twister instance, whereas our implementation using OpenMP combines two single-precision random numbers from two different Mersenne Twister instances. The difference in the implementations results from the circumstance that on the part of our implementation for the GPU, each CUDA thread runs its own Mersenne Twister without interacting with other CUDA threads, whereas on the part of our implementation for the CPU, we use 4 threads, each processing a large pool of Mersenne Twisters, that are able to combine random numbers from different Mersenne Twister instances.

Execution times

Table 4.2 lists the mean execution times in nanoseconds per random number $\bar{t}_{\text{sample}}$ as well as the number of samples per second $\bar{r}_{\text{sample}}$. Both the GPU and the CPU ran 3840 Mersenne Twister instances, where each of them was to generate 6×2560 random numbers. All measurements were repeated 10 times.

	Nvidia Tesla C1060		Intel Core i7-920	
precision	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$
single	0.17	5.88	1.11	0.90
double	0.56	1.79	2.26	0.44

Table 4.2.: Mean execution times in nanoseconds per random number $\bar{t}_{\text{sample}}$ as well as the number of samples per second $\bar{r}_{\text{sample}}$. All measurements were repeated 10 times.

To achieve the execution times listed in Table 4.2, we chose the order in which random numbers are stored in the graphics card's main memory different from the order in which random numbers are stored in the main memory of the host. As a consequence, random number sequences on the GPU will not be the same as random number sequences on the CPU when reading them successively from main memory during a simulation. However, the quality of the random number sequences should be unaffected.

4.3. An alternative approach—CDAran32 and CDAran64

As can be seen from Table 4.1 and 4.2, mean execution times to generate a single random number differ by up to one order of magnitude, when comparing Ranlux with Mersenne Twister. It therefore seems natural to consider methods that allow to generate random numbers even faster than by means of the Mersenne Twister algorithm.

We thus spent some time in creating a generator that is optimized for Nvidia graphics cards. To make this generator perform well on the GPU, we considered it necessary to use only simple integer arithmetic, and to avoid conditional clauses that interrupt the working flow of the CUDA threads. On the part of the single-precision version of this generator, hereafter referred to as CDAran32, we directly ran into some trouble due to the rather limited number of 'recommended' 32-bit generators. Approaches for generating random numbers when being restricted to 32-bit, as is principally the case when doing GPGPU with current graphics hardware,[4] are given in [22]. The authors point out that neither of them passes serious random number tests, but combining them may provide acceptable statistical properties. We therefore combined a 32-bit-XOR-shift generator (state y and parameters (b_1, b_2, b_3)) with a multiply-with-carry generator

[4]Current Nvidia graphics cards are able to perform calculations with double-precision, but their theoretical peak performance in so doing is much smaller than their theoretical single-precision peak performance (see Table 3.1).

(state z and parameter a) as follows

$$\left.\begin{aligned} y &\leftarrow y \wedge (y \gg b_1)\,, \\ y &\leftarrow y \wedge (y \ll b_2)\,, \\ y &\leftarrow y \wedge (y \gg b_3)\,, \\ z &\leftarrow a(z \,\&\, [2^{16}-1]) + (z \gg 16)\,, \\ \hline \text{random sample:}\quad x &\leftarrow 2.3283064365386962\text{E-}10 \times (y \wedge z) \end{aligned}\right\} \tag{4.5}$$

Using the parameters given in [22], a total of 90 different 4-tupels (b_1, b_2, b_3, a) can be composed. These 4-tupels, from now onwards called 'configurations', then are distributed over a set of 7680 generators, all of type (4.5), that is, each of the 7680 generators receives a number $\in [0, 89]$ at random, which relates it to a certain configuration. Having initialized all 7680 generator instances, and after having assigned start values to y and z, as prescribed in [22], each generator is mapped onto a CUDA thread that, whenever the CDAran32 kernel executes, loads its configuration into local memory, and a randomly but bijectively chosen state (y, z) into shared memory. Then the idea is to organize the 7680 random number generators into groups, and to make all CUDA threads within the same group interchange their states each time after having generated a fixed amount of random numbers, say, 8, for instance. The latter step then is repeated until all requested random numbers are generated.

Algorithm 3 summarizes the CDAran32 algorithm in some kind of structure chart.

CDAran32 algorithm

```
copy configuration into local memory
copy randomly but bijectively chosen state (y, z) into shared memory
synchronize with CUDA threads within the same thread block
for i = 0 to number of requested random samples do
  for j = 0 to 8 do
    x ← process update-kernel, implementing Eq. (4.5)
    write random number x to the output
  end for
  synchronize with CUDA threads within the same thread block
  interchange states (y, z) within the same thread block at random
  synchronize with CUDA threads within the same thread block
  i ← i + 8
end for
copy state (y, z) back to device memory
```

Algorithm 3: Structure chart for the CDAran32 algorithm.

Algorithm 3 describes the acting of a single CUDA thread which corresponds to an

appropriate kernel. Our implementation that realizes this algorithm is given by Listing D.2 in Appendix D.2.

The double-precision version of CDAran32, hereafter referred to as CDAran64, is based on the same idea, but in contrast to the double-precision version of Ranlux or Mersenne Twister, it uses native double-precision algorithms à la Numerical Recipes [22].

Execution times

Since both CDAran32 and CDAran64 are designed to perform well on Nvidia graphics hardware, there are no implementations for the CPU. For this reason, Table 4.3 lists only execution times for the Tesla C1060 and the GeForce GTX285 graphics card. Each of the 7680 CDAran32/64 instances was to generate a set of 3×2560 random numbers. All measurements were repeated 10 times.

	Nvidia Tesla C1060		Nvidia GeForce GTX285 @ default clock	
precision	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$	$\bar{t}_{\text{sample}}$ in ns	$\bar{r}_{\text{sample}}/10^9$
single	0.094	10.64	0.057	17.54
double	0.257	3.89	0.187	5.35

Table 4.3.: Mean execution times in nanoseconds per random number $\bar{t}_{\text{sample}}$ as well as the number of samples per second $\bar{r}_{\text{sample}}$. All measurements were repeated 10 times.

As given by Table 4.3, producing random numbers with CDAran32/64 is almost twice as fast as producing random numbers with Mersenne Twister.

To investigate the quality of CDAran32/64's random number sequences, we simulated the (exactly solveable) two-dimensional Ising model in zero magnetic field by means of the single-cluster algorithm, which is extremely sensitive to subtle correlations within random number sequences. Information on testing random number generators are given in Sections 5.2.2 and E.4.[5] As will be shown there, Monte Carlo estimates of the internal energy per spin and the specific heat per spin on (16×16) and (32×32) squared lattices are compatible with exact calculations, within errors. We therefore deem it justified to use the CDAran32/64 random number generator for parallel Monte Carlo simulations of Ising-like spin models.

[5] Similar investigations were already presented by Ferrenberg and Landau [10].

5. Simulating the Ising model on parallel computer architectures

While the previous chapters were primarily concerned with theoretical aspects of certain spin models as well as with basics about simulating them on the computer by means of Monte Carlo methods, the subject of the present chapter is on how to implement the Ising model on Nvidia graphic cards and multi-core CPUs respectively, and on simulating the critical Ising model.

There are some practical reasons for focussing on the Ising model. First, the Ising model is exactly solveable in two dimensions and zero external magnetic field, so that simulation results, obtained from our implementations, can be compared against exact calculations, which in turn allows to check for deficiencies in our codes. By comparing Monte Carlo estimates from simulating the two-dimensional Ising model in zero external magnetic field by means of the single-cluster algorithm against exact calculations, we are also able to check for the quality of our parallel random number generators. Since subtle correlations within the random number sequences may influence the single-cluster algorithm in such a way that it generates clusters that are systematically too large or too small [10], simulation results also show systematic deviations from exact calculations. Second, the Ising model is the simplest instance of a large class of Ising-like spin models. An implementation that allows to simulate the Ising model, also allows to simulate any other Ising-like spin model by just doing simple modifications of the source code. In Section 6, we will study the Ising spin glass, which basically differs from the Ising model in such a way that both ferromagnetic and anti-ferromagnetic spin-spin interactions occur instead of either the one or the other, as is the case in the Ising model.

For reasons of simplicity, we first simulate the Ising model by means of the Metropolis algorithm. Due to its locality, the Metropolis algorithm seems to be an obvious way to map the Ising model (and also other spin models) onto parallel computer architectures. Although the Metropolis algorithm suffers from critical slowing down, its field of application is extremely wide,[1] which in its own right is reason enough to continually check for hardware that allows to apply it to (spin) models with larger and larger extent, with the time required for simulating these models being only a fraction of the time required by current computers. By simulating the Ising model by means of the Metropolis algorithm on Nvidia Tesla C1060 graphics cards and on an Intel Core i7-920, as representative of current x86 quad-core CPUs, we aim to get reliable information about the gain in performance that can be achieve when simulating Ising-like spin models on GPUs instead

[1] In Section 6, for instance, we will study the Ising spin glass by means of the Metropolis algorithm. Since efficient cluster algorithms for this kind of model are not available, the Metropolis algorithm is the preferred choice.

of CPUs.

We also consider cluster algorithms for possible implementations by means of CUDA and OpenMP. Even if porting them to CUDA could not be accomplished successfully, we will use a CPU implementation of the single-cluster algorithm to check for subtle correlations within the random number sequences of our parallel random number generators [10], and also to investigate the critical behavior of the Ising model by means of finite size scaling methods. Although the latter point is not directly related to this work, we decided to see beyond the scope of this thesis in order to get an insight into methodologies commonly used in this research field [8, 20].

5.1. Simulations based on the Metropolis algorithm

The Metropolis algorithm is one of the most established Monte Carlo algorithms. In addition to its simplicity, it is well suited for parallel implementations since its locality allows to concurrently operate on different lattice sites. As mentioned in Section 3.1, processing, for instance, the Metropolis algorithm on parallel computer architectures requires to take care of memory access patterns, data decomposition, and synchronization of threads/processes.

To stage the Metropolis algorithm on graphics cards and multi-core CPUs, we use the so-called *checkerboard decomposition*, which divides the entire lattice into even and odd sites, where a lattice site is said to be even if its cartesian coordinates add up to an even value. Otherwise it is said to be odd. The procedure then is to first *update all spins on even lattice sites*, and afterwards to *update all spins on odd lattice sites* in a second step, where updating a spin means to use the Metropolis acceptance ratio (2.10) for changing the spin's orientation. After having performed both of the update steps, defining a so-called *sweep*, all spins were taking into account to change their orientation. Subsequently, we denote this way to update the entire lattice as 'checkerboard procedure', illustrated in Figure 5.1.

The necessity to decompose the lattice into disjoint sets of spins, here spins on even and odd lattice sites, follows from the circumstance that we aim to update spins in

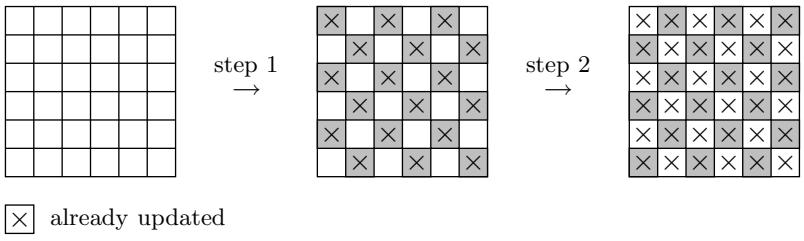

Figure 5.1.: Checkerboard procedure applied to the two-dimensional Ising model on a squared lattice. Spins are identified with boxes, where the darkened ones refer to spins that have been updated during the corresponding update step. Boxes with crosses inside refer to spins that have already been updated. After having performed the second step, all spins have been updated.

parallel, where the chronological order in which these updates are actually performed is decided by the hardware at runtime—at least on MIMD and SIMT machines, such as our devices—. Since each Metropolis step requires to evaluate the nearest-neighbor sum $\sum_{\langle ij \rangle} s_i s_j$, spins on odd lattice sites are not allowed to change their orientation while spins on even lattice sites are updated, and vice versa. In fact, this is the acting of the checkerboard procedure, which thus prevents race conditions.

5.1.1. Remarks on implementing the checkerboard procedure

Since updating spins on even lattice sites is completely independent from updating spins on odd lattice sites, it appears to be most natural to map all spins that are updated during either the first or the second step of the checkerboard procedure (see Figure 5.1) onto a certain number of threads which then execute in parallel.

On the part of the Intel Core i7-920, the simplest approach is to divide the entire lattice into 4 sublattices, each being processed by a single thread which applies the checkerboard procedure to its sublattice. Each time after having finished either the first or the second update step, all threads need to synchronize with each other in order to assure theirselves that all threads that process adjacent sublattices are done. Consecutive application of this procedure then generates a Markov chain of states, system quantities can be extracted from. Figure 5.2 schematically illustrates the acting of the Core i7-920 quad-core CPU in consecutively applying the checkerboard procedure for the Ising model to an (8×8) squared lattice with periodic boundary conditions. Each of the four threads

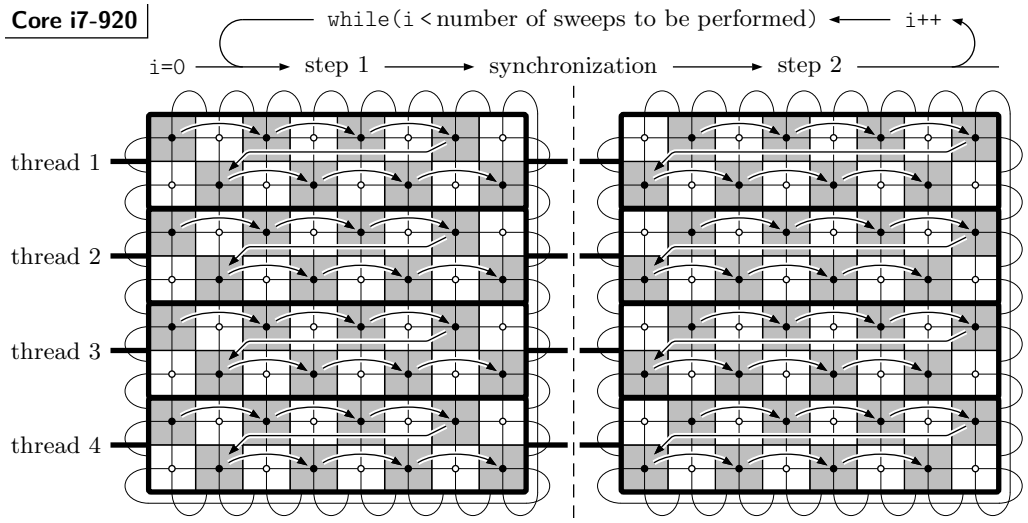

Figure 5.2.: Schematically illustration of the acting of the Core i7-920 quad-core CPU in consecutively applying the checkerboard procedure for the two-dimensional Ising model to an (8×8) squared lattice. Arrows within the lattices indicate the order in which spins are update by their respective threads. The meaning of white and dark boxes is the same as in Figure 5.1. Solid circles refer to spins that are actually updated during the respective update step, while open circles refer to nearest-neighbor spins that are required for the nearest-neighbor sum.

executes on one of the four physical execution units (cores) of the Core i7-920. Since all threads have access to the same main memory—they also share the CPU's L3 cache—, exchanging borders of domains is not necessary, as would be the case when processing them on different machines by means of MPI, for instance. Using the CPU's SSE units, each thread is able to operate 4 spins at the same time, that is, all even/odd spins in the same row of our (8×8) lattice (see Figure 5.2) can be updated simultaneously in a SIMD-like manner.

As detailed in Chapter 3, graphics cards require the number of threads to exceed the number of physical execution units available on the device in order to compensate for non-cached main memory requests. For this reason it is not far to assign a single CUDA thread to each spin that is updated during the respective update steps of the checkerboard procedure. Since the number of CUDA threads that are required to concurrently update half the lattice in each of the two update steps rapidly increases with the extent

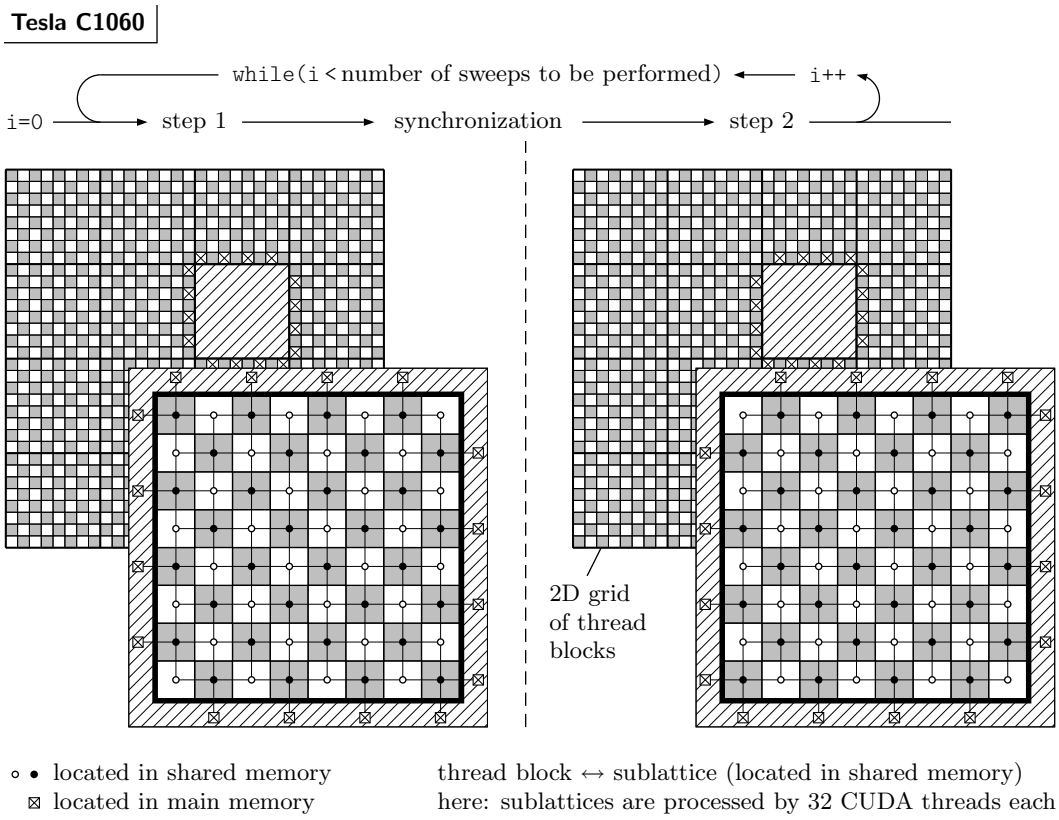

Figure 5.3.: Schematically illustration of the acting of the Nvidia Tesla C1060 in consecutively applying the checkerboard procedure for the two-dimensional Ising model to a (32×32) squared lattice. The entire lattice is divided into sublattices which then are mapped onto thread blocks, where each of the sublattices is concurrently processed by 32 CUDA threads. The meaning of white and dark boxes is the same as in Figure 5.1. Solid circles denote spins that are actually updated during the respective update step. Open circles and boxes with crosses inside refer to nearest-neighbor spins that are required for the nearest-neighbor sum.

of the lattice, it becomes necessary to divide the entire lattice into sublattices which then are mapped onto thread blocks of up to 512 CUDA threads each. Figure 5.3 illustrates the acting of a total of 512 CUDA threads, processing the checkerboard procedure for the two-dimensional Ising model on a (32×32) squared lattice with periodic boundary conditions. If we would have mapped the whole lattice onto a single thread block, only one of the 30 multiprocessors of the Tesla C1060 would be busy while the other 29 would do nothing. How to actually choose the geometry of the sublattices strongly depends on the geometry of the entire lattice, where the programmer is to ensure that the number of sublattices (thread blocks) is at least of the order of available multiprocessors.

To have the CUDA code for simulating the Ising model as fast as possible, we use the shared memory of the GPU in such a way that entire sublattices are loaded into it when being processed by CUDA threads. There are two reasons for loading entire sublattices into shared memory:

1. Memory requests, due to loading sublattices from main memory, occur only once, and moreover they are aligned and coalesced, that is, memory transfer rates should be high (see performance issues given in Appendix C.3).
2. Nearest neighbor relations are the same as for the entire lattice, which allows to access nearest-neighbor spins in the shared memory as well as in the main memory in a straightforward way.

In our implementations of the checkerboard procedure (for the two-dimensional and the three-dimensional Ising model), we do not load boundary spins (in Figure 5.3 these spins are referred to as boxes with crosses inside) into shared memory since requesting for them may lead to uncoalesced memory accesses (low memory transfer rates). In addition to that, boundary spins enter the computation of the corresponding nearest-neighbor sums only once, which makes it unpractical to first load them into shared memory, and then to read them from shared memory in order to use them for just one calculation. There are some further aspects that need to be considered for having both the GPU and the CPU perform on their best. We used the following performance issues:

1. Our implementations precompute the acceptance ratios $A(\mu \to \nu)$. If there is no external magnetic field, as is the case in our simulations, the energy difference due to flipping a certain spin, say, s_i is given by

$$\begin{aligned}(E_\nu - E_\mu)_i &= -J \left[s_i^{(\nu)} \sum_{j=\text{neighbor}(i)} s_j^{(\nu)} \quad - \quad s_i^{(\mu)} \sum_{j=\text{neighbor}(i)} s_j^{(\mu)} \right] \\ &= -J \left[-s_i^{(\mu)} \sum_{j=\text{neighbor}(i)} s_j^{(\mu)} \quad - \quad s_i^{(\mu)} \sum_{j=\text{neighbor}(i)} s_j^{(\mu)} \right] = 2J s_i^{(\mu)} \sum_{j=\text{neighbor}(i)} s_j^{(\mu)} .\end{aligned}$$

   For $x = (s_i^{(\mu)} \sum_j s_j^{(\mu)}) > 0$, the corresponding acceptance ratios $A(\mu \to \nu) = \mathrm{e}^{-\beta 2Jx}$ are stored in main memory or constant memory to request for them during the simulation. $x \leq 0$ corresponds to $A(\mu \to \nu) = 1$, that is, the spin s_i will be flipped.
2. Our implementations generate large sets of random numbers in order to achieve the mean execution times given by Tables 4.1-4.3.

We created implementations of the checkerboard procedure for the two-dimensional and the three-dimensional Ising model using CUDA and OpenMP respectively. Listing E.1 in Appendix E.1 gives the source code of our CUDA kernel that realizes the first update step of the checkerboard procedure for the two-dimensional Ising model.

5.1.2. Measured quantities

Over the course of simulating the Ising model by consecutively updating the system's spins by means of the checkerboard procedure, measurements are taken every n-th Monte Carlo sweep in order to derive system properties. All quantities introduced in Section 1.2 can be deduced from time series of the sum over the system's spin values $\sum_i s_i$, and the sum over the product of nearest-neighbor spins $\sum_{\langle ij \rangle} s_i s_j$. Our implementations therefore include functions (kernels) that evaluate these sums for states $\{\mu_1, \ldots, \mu_{\tilde{M}}\}$ which are separated by a certain number of Monte Carlo sweeps. Deducing quantities like the internal energy, the specific heat, the magnetic susceptibility or the absolute magnetization then is straightforward.

The internal energy calculates as the mean value of the estimates $\{E_{\mu_n}\}$ which result from evaluating Eq. (1.5) for spin configurations $\{\mu_n\}$. The Monte Carlo estimator for the internal energy per spin E_{MC} then reads

$$E_{\mathrm{MC}} = \frac{1}{N}\left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}} E_{\mu_n}\right), \qquad E_{\mu_n} = -J\sum_{\langle ij \rangle} s_i^{(\mu_n)} s_j^{(\mu_n)} - h\sum_i s_i^{(\mu_n)}, \tag{5.1}$$

where N is the number of lattice sites, and $s_i^{(\mu_n)}$ refers to the value of spin s_i in state μ_n. For later convenience we set $J = 1$, $h = 0$ and $k_{\mathrm{B}} = 1$. Equation (5.1) directly follows from what was figured out in Section 2.1.

The Monte Carlo estimator for the specific heat per spin $(c_V)_{\mathrm{MC}}$ is

$$(c_V)_{\mathrm{MC}} = \frac{\beta^2}{N}\left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}} E_{\mu_n}^2 - \left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}} E_{\mu_n}\right)^2\right). \tag{5.2}$$

Investigating the mean magnetization per spin $\langle m \rangle = N^{-1}\langle M \rangle$ is somewhat more difficult, since in the limit of an infinitely long period of time, $\langle M \rangle$ vanishes even for $T < T_c$, as detailed in Section 1.2. In order to account for the existence of the two symmetry-equivalent ground states of the Hamiltonian (1.5), one usually considers the absolute magnetization per spin $\langle |m| \rangle = N^{-1}\langle |M| \rangle$, whose Monte Carlo estimator is given by

$$|m|_{\mathrm{MC}} = \frac{1}{N}\left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}} |M_{\mu_n}|\right) = \frac{1}{N}\left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}} \Big|\sum_i s_i^{(\mu_n)}\Big|\right). \tag{5.3}$$

In fact, this quantity gives the expected behavior for temperatures $T < T_c$, but averaging over the absolute values of the estimates $\{M_{\mu_n}\}$ also results in an asymptotic small but non-zero mean magnetization for $T \gtrapprox T_c$. Only in the thermodynamic limit this quantity

is expected to tend to zero for temperatures larger than T_c.

To estimate the magnetic susceptibility per spin, we incorporate the theoretical result $\langle M \rangle = 0$, that is,

$$\chi = \frac{\beta}{N}\Big(\langle M^2 \rangle - \langle M \rangle^2\Big) = \frac{\beta}{N}\langle M^2 \rangle \,.$$

The Monte Carlo estimator for the magnetic susceptibility per spin χ_{MC} then reads

$$\chi_{\mathrm{MC}} = \frac{\beta}{N}\left(\frac{1}{\tilde{M}}\sum_{n=1}^{\tilde{M}}\left(\sum_i s_i^{(\mu_n)}\right)^2\right). \tag{5.4}$$

5.1.3. Simulation results and execution times

To investigate the Ising model in zero external magnetic field, we first determined the number of Monte Carlo sweeps that are necessary to evolve the system to thermal equilibrium. We therefore considered a rather short set of Monte Carlo estimates of the internal energy. The system is said to be thermalized if its properties only slightly fluctuate around some constant values. For lattices with periodic boundary conditions and extents $L = \{16, 32, 64, 128, 256\}$ in two dimensions and $L = \{16, 24, 32, 48, 64\}$ in three dimensions, thermal equilibrium was reached after having performed about 1000 Monte Carlo sweeps.

Second, for all lattice extents listed above, we determined the autocorrelation time of the internal energy τ_E to know about how long the system has to be approximately evaluated until it reaches a state that significantly differs from the state that was previously used to take measurements.[2]

We then set up our simulations to perform 10000 Monte Carlo sweeps—to be on the safe side—to reach thermal equilibrium, and from that point on to take measurements using spin configurations that are separated by τ_E Monte Carlo sweeps. In this way, we simulated the two-dimensional and the three-dimensional Ising model for different lattice extents L and different random number generators. In particular a total of 10 replica, each containing $1000000 = 10^6$ Monte Carlo estimates of E, c_V, $|m|$ and χ, was generated for each combination (L, random number generator). All measurements were taken at the critical point, which for the two-dimensional Ising model is given by the inverse critical temperature

$$\beta_c = \frac{1}{2}\ln(1+\sqrt{2}) = 0.4406867935\ldots \tag{5.5}$$

In the case of the three-dimensional Ising model, we used $\beta_c \approx 0.22165$ [2].

Figure 5.4 displays deviations ΔE and Δc_V between exact calculations and Monte Carlo estimates of the internal energy per spin and the specific heat per spin in the case of the two-dimensional Ising model. All quantities were evaluated by means of the `UWerr`

[2] We considered the internal energy due to its rather small autocorrelation time, compared to that of c_V, $|m|$ and χ. Separating consecutive measurements by τ_E Monte Carlo sweeps then leads to a negligible loss in information for quantities E, c_V, $|m|$ and χ, due to skipping $\tau_E - 1$ intermediate configurations.

script, described in Section 2.4.

Simulation results for the two-dimensional and the three-dimensional Ising model in

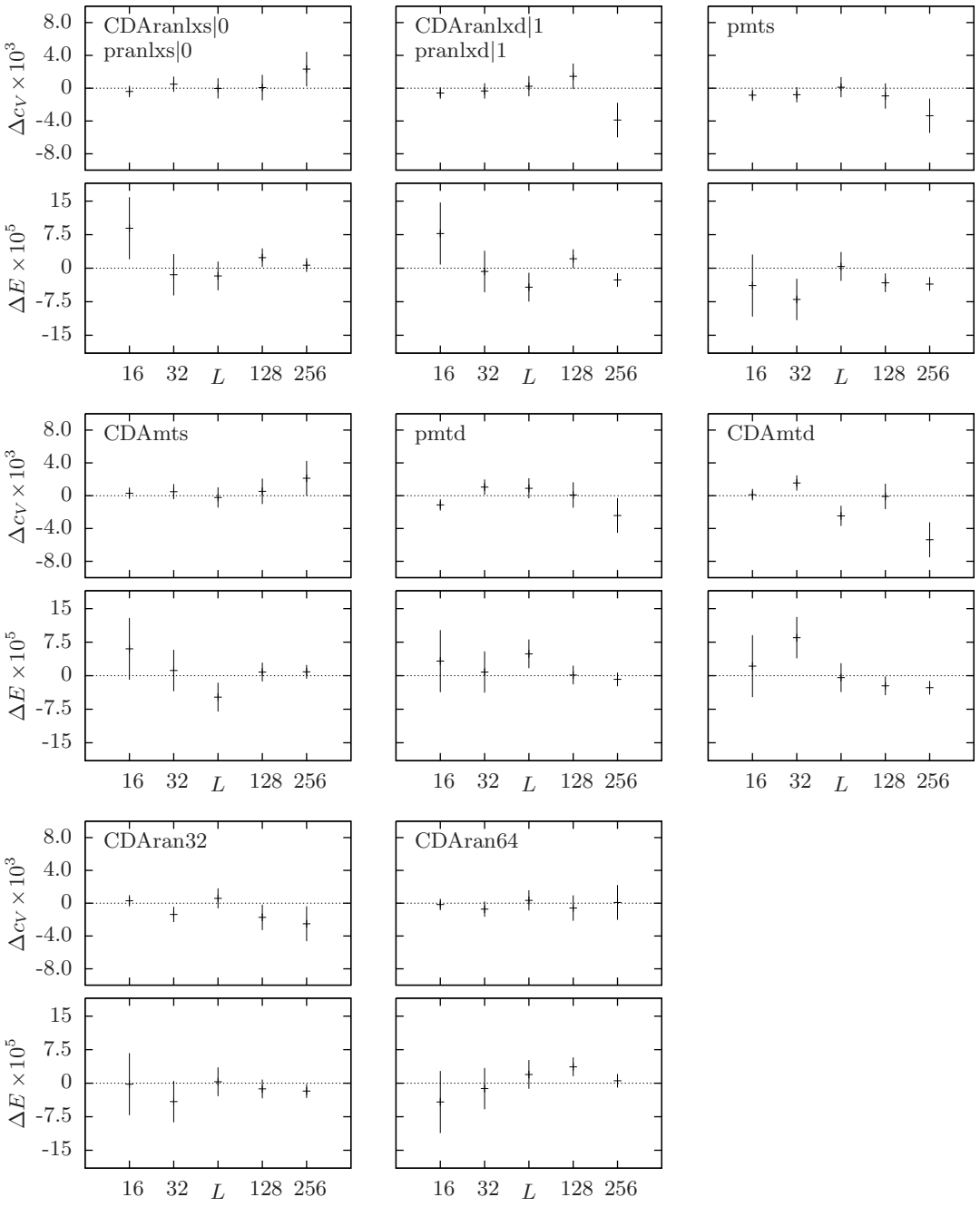

Figure 5.4.: Deviations between exact calculations and Monte Carlo estimates of the internal energy per spin and specific heat per spin in the case of the two-dimensional Ising model, simulated by means of the Metropolis algorithm.

zero external magnetic field are listed in Tables E.1-E.4 in Appendix E.2. All random number generators that were used yield Monte Carlo estimates that are compatible with exact calculations within errors as well as with literature values [2, 25].

Since our implementations of the Ranlux random number generator produce exactly the same random number sequences on both the GPU and the CPU—this it not the case for the other random number generators used, as noted in Chapter 4—, simulation results are also exactly the same. Figure 5.4 thus merges the corresponding deviations each into the same plot. In a certain sense, this circumstance also attests the correctness of our implementations, at least in the meaning that simulations on both the GPU and the CPU give exactly the same output when using Ranlux, which in turn also gives first information on the reliability of GPGPU computations.

For later convenience, and as already used in Figure 5.4, we introduce the following abbreviations for our random number generators:

abbreviation	random number generator
C1	'pranlxs|x'—single-precision version of the Ranlux random number generator using OpenMP to perform on multi-core CPUs. 'x' specifies the luxury level.
C2	'pranlxd|x'—double-precision version of the Ranlux random number generator using OpenMP to perform on multi-core CPUs. 'x' specifies the luxury level.
C3	'pmts'—single-precision version of the Mersenne Twister random number generator using OpenMP to perform on multi-core CPUs.
C4	'pmtd'—double-precision version of the Mersenne Twister random number generator using OpenMP to perform on multi-core CPUs.
G1	'CDAranlxs|x'—single-precision version of the Ranlux random number generator using CUDA. 'x' refers to the luxury level.
G2	'CDAranlxd|x'—double-precision version of the Ranlux random number generator using CUDA. 'x' refers to the luxury level.
G3	'CDAmts'—single-precision version of the Mersenne Twister random number generator using CUDA.
G4	'CDAmtd'—double-precision version of the Mersenne Twister random number generator using CUDA.
G5	'CDAran32' random number generator using CUDA.
G6	'CDAran64' random number generator using CUDA.

Table 5.1.: Abbreviations for random number generators used in this diploma thesis.

Up to now, we simulated the Ising model on squared lattices with maximum extent $L = 256$ as well as on cubic lattices with maximum extent $L = 64$. Although investigating

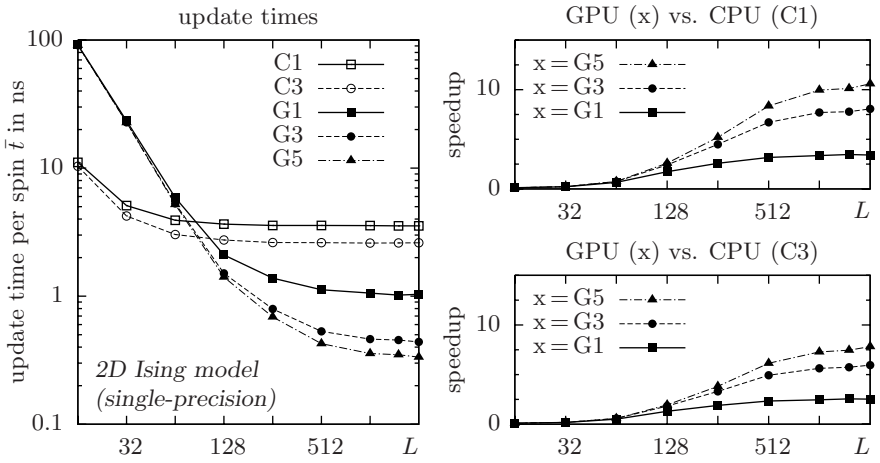

Figure 5.5.: Mean update times $\bar{t}$ per spin, and speedups for simulating the two-dimensional Ising model on squared lattices with extents L by means of the Metropolis algorithm (checkerboard procedure). For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920.

the Ising model on lattices with extents much larger than the present ones can be more efficiently done by means of cluster algorithms, due to critical slowing down, there are models for which the Metropolis algorithm is the preferred choice. For this reason, we measured execution times for simulating the Ising model on squared/cubic lattices with periodic boundary conditions and extents up to $L = 2048$ in two dimensions, and extents up to $L = 256$ in three dimensions. We then calculated mean update times per spin.[3]

To do so, we set up our simulations to take measurements from spin configurations that are separated by $n = 20$ Monte Carlo sweeps, which seems to be justified for small lattices, but might be undervalued for larger ones. Since simulations on lattices with large extents will perform a little faster if $n > 20$, our estimates of the mean update times per spin will not suffer from the choice $n = 20$. To obtain halfway reproducible execution (update) times, the overall number of performed Monte Carlo sweeps N was chosen to be much larger than n. Figures 5.5 and 5.6 summarize the mean update times per spin for simulating the Ising model on squared/cubic lattices with extents L using single-precision random numbers. In addition, speedups are given. Execution times were measured for Nvidia Tesla C1060 graphics cards and for an Intel Core i7-920.

For simulating the Ising model on lattices with large extents, there is a decrease in the update times per spin of up to one order of magnitude when using the Tesla C1060 instead of the Core i7-920, that is, the graphics card performs much faster than our quad-core CPU. Changing to lattices with much smaller extents, quite the reverse is true. The latter point follows from a very poor workload of the graphics card in these cases. To put it simple, when simulating the Ising model on lattices with small extents, the number of required CUDA threads is insufficient to have all the GPU's execution units not run idle.

Although not explicitly depicted, our simulations that use double-precision random

[3]The update times per spin also include the time to produce random numbers and to take measurements.

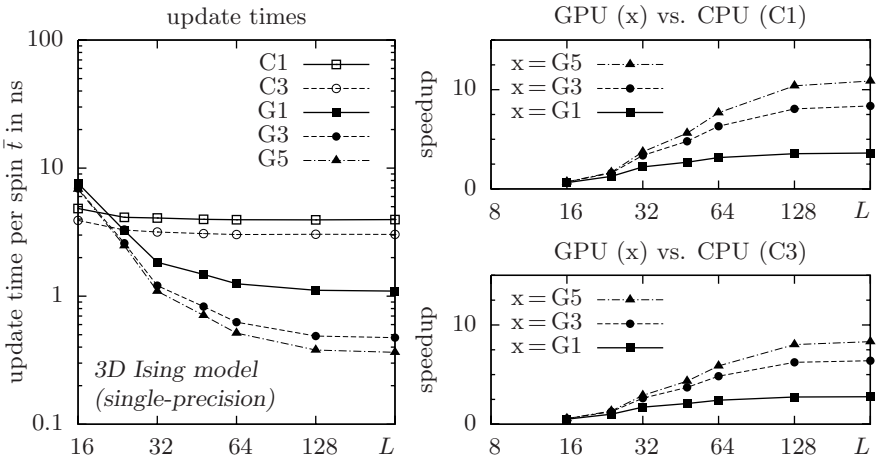

Figure 5.6.: Mean update times $\bar{t}$ per spin, and speedups for simulating the three-dimensional Ising model on cubic lattices with extents L by means of the Metropolis algorithm (checkerboard procedure). For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920.

numbers give update times that are larger than their single-precision equivalents by a factor $1.5 - 2.0$, except for simulations that use the Ranlux random number generator which in its double-precision version is principally 'more luxury'. To support what was said, Table E.5 in Appendix E.3 lists the total per-replicum execution times of all our simulations of the Ising model as well as the corresponding mean update times per spin.

Summarizingly, we conclude that the Tesla C1060 is able to outperform current quad-core CPUs when simulating, for instance, the Ising model on lattices with large extents, whereas simulating this model on lattices with small extents will make the graphics card perform below its best. Even, program execution times strongly depend on the random number generator used. While CDAran32/64 and the Mersenne Twister random number generator are well suited for parallel Monte Carlo simulations of Ising-like spin models using CUDA, generating random numbers with Ranlux becomes the bottleneck of the simulation. We thus deem Ranlux not to be the preferred choice for such simulations. This point may change when performing, for instance, QCD simulations on the GPU, due to a larger number of arithmetic operations per random number.

Comparing our maximum speedups for 'Nvidia Tesla C1060 vs. Intel Core i7-920', which are about $7 - 10$, with speedups presented in [21], there is a discrepancy of one order of magnitude. The authors state a speedup of about 60 in the two-dimensional case, and a speedup of about 35 in the three-dimensional case, where both of their implementations using CUDA give mean update times per spin that are significantly larger than ours—we determined a factor of about 1.5—. That is, the authors obtained speedups that are larger than ours by a factor $6 - 8$ although their graphics card programs perform slower than ours by a factor 1.5. In addition, the authors used Nvidia GeForce GTX280 consumer graphics cards,[4] which are not suitable for high-performance

[4]We compared the performance of Nvidia GeForce GTX285 consumer graphics cards with the performance of professional Nvidia Tesla C1060 graphics cards. In the case of the Ising model, the GTX285

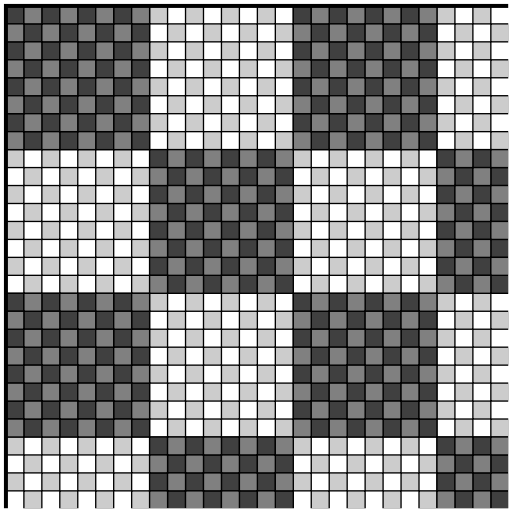

Figure 5.7.: Double-checkerboard decomposition of a two-dimensional lattice. The entire lattice is divided into sublattices, where the darkened ones denote even sublattices and the remaining ones denote odd sublattices. The idea is to consecutively update even and odd sublattices, each multiple times in succession, using the standard checkerboard procedure.

computing, as will be discussed in Chapter 7. As a consequence, performance ratings in [21] are absolutely missleading.

5.1.4. Improved checkerboard procedure

To achieve lower update times per spin, an obvious improvement of our current implementations might be to update fractions of the lattice multiple times in succession by means of the checkerboard procedure. The idea is to keep sublattices in local memory spaces for a certain time, and to alter their spin configurations significantly without requesting for them again and again, as is the case in our current implementations.

Implementing this procedure by means of a parallel programming approach requires to divide the entire lattice into even and odd sublattices in order to avoid race conditions when requesting for spins around the edges of the sublattices (see Figure 5.7). Updating the entire lattice then means to first update all even sublattices multiple times using the checkerboard procedure within each of the sublattices, and afterwards to update all odd sublattices the same way. We refer to this procedure as *double-checkerboard procedure*.

Due to dividing the entire lattice into even and odd sublattices, spins around the edges of the presently updated sublattices will not change, that is, the double-checkerboard procedure introduces *Dirichlet boundary conditions* during the 'multi-updates'. In spite

performed about 1.5 times faster than the Tesla C1060. We observed similar differences in performance for producing random numbers with the CDAran32/64 random number generator (see Section 4.3, Table 4.3). Here the GTX285 performed more than 1.6 times faster than the Tesla C1060. These discrepancies follow from the reduced memory bandwidth of the Tesla C1060, compared to the GTX285 (see Table 3.1), which directly influences the performance of the Tesla C1060.

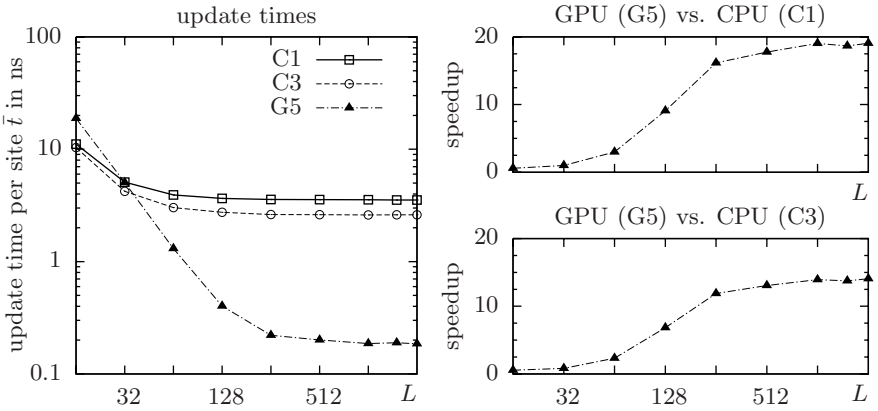

Figure 5.8.: Mean update times $\bar{t}$ per spin, and speedups for simulating the two-dimensional Ising model by means of the double-checkerboard procedure on squared lattices with periodic boundary conditions and extents L. The number of multi-updates m was set to $m = n = 20$. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

of this modification, the algorithm obviously still satisfies the condition of ergodicity and the condition of detailed balance.

Skipping details of the implementation of the double-checkerboard procedure—we created an implementation for the two-dimensional Ising model using CUDA—, Figure 5.8 summarizes the mean update times per spin on a Tesla C1060, where the `CDAran32` random number generator was used. Similar to the runtime measurements in Section 5.1.3, the number of Monte Carlo sweeps that separate consecutive measurements was set to $n = 20$, and the total number of Monte Carlo sweeps N was chosen to be much larger than n. The number of multi-updates m was set to $m = n = 20$.

As can be seen from Figure 5.8, the mean update times per spin decrease by almost a factor 2, which leads to speedups that are nearly twice as large as those given in Figure 5.5. Unfortunately, this is only half the story. Concurrently to the decrease in the update times, the autocorrelation times dramatically increase with the number of multi-updates. We therefore investigated the dependency of the autocorrelation time of the magnetic susceptibility from the number of multi-updates m. For this purpose, simulations of the two-dimensional Ising model on squared lattices with extents $L = \{16, 32, 64, 128\}$ were performed, where the number of multi-updates was varied over $m \in \{1, 2, 5, 10, 20\}$. The geometry of the sublattices was chosen to be (8×8) in the case of lattice geometry (16×16), and (16×8) in all other cases. Figure 5.9 contrasts the decrease in the mean update times per spin as m grows, and also the increase in the autocorrelation times in so doing. The plots on the right hand side give the gain in performance when comparing the double-checkerboard procedure with the standard checkerboard procedure. The gain calculates as

$$\text{gain} = \frac{\bar{t}_{\text{standardCheckerboard}}}{\bar{t}_{\text{doubleCheckerboard}}} \Big/ \frac{(\tau_{\text{int,doubleCheckerboard}})_\chi}{(\tau_{\text{int,standardCheckerboard}})_\chi},$$

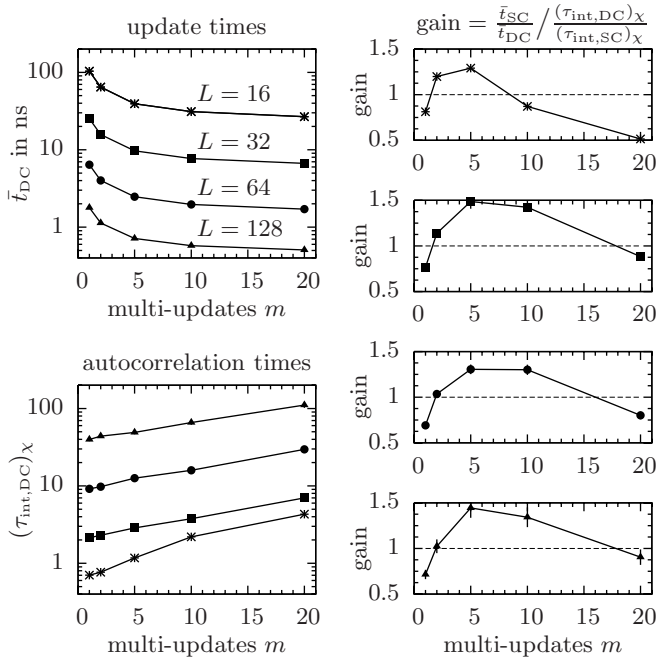

Figure 5.9.: Mean update times $\bar{t}_{\mathrm{DC}}$ per spin and autocorrelations times τ_χ as a function of the number of multi-updates m. The gain in performance, compared to the standard checkerboard procedure, is illustrated by the plots on the right hand side.

with $\bar{t}_{\mathrm{standardCheckerboard}}$ and $\tau_{\mathrm{int,standardCheckerboard}}$ from simulations that were performed by means of the standard checkerboard procedure. We used the term 'DC' to abbreviate 'double-checkerboard procedure', and the term 'SC' to abbreviate 'standard checkerboard procedure'.

Simulating the two-dimensional Ising model by means of the double-checkerboard procedure lowers the mean update times per spin significantly. On the other hand it leads to an increase of the autocorrelation times, which grow exponentially with the number of multi-updates m. All in all, the increase in the autocorrelation times compensate for the lowering of the mean update times per spin, which then results in gain-values smaller than 1 if $m \gg 10$. Best performance, compared to the standard checkerboard procedure, can be achieved when sublattices are updated about 5 times in succession. Since similar optimizations also need to be done on the part of the CPU implementations, speedups given in Figures 5.5 and 5.6 will not change significantly.

Why do the autocorrelation times increase with the number of multi-updates?

As known from chapter 1, at criticality there are clusters of all sizes up to the lattice extent, corresponding to effective long-range interactions. By performing multi-updates of subdomains, as detailed in this subsection, information about the system's evolution propagate only within these subdomains, at least during the multi-updates. As a consequence, the time (in units of Monte Carlo sweeps) that is necessary to propagate

information over the entire lattice increases with m, and autocorrelation times grow.

5.2. Simulations based on cluster algorithms

As mentioned several times before this section, the critical behavior of certain spin models (e.g. the Ising model) can be studied more efficiently by means of cluster algorithms, due to suppressed critical slowing down. Subsequently, we give some remarks on implementing cluster algorithms in parallel. In addition, some measurements were taken to compute the critical exponents of the two- and the three-dimensional Ising model.

5.2.1. The Swendsen-Wang cluster algorithm and CUDA

At its core the single-cluster algorithms differs from the Swendsen-Wang cluster algorithm in just looking for one cluster per iteration instead of dividing the entire lattice into clusters and to flip half of them on average. As far as we know, efficient implementations of the single-cluster algorithm using multiple processors are extremely rarely. Spending time on trying to create an implementation of the single-cluster algorithm using CUDA thus seems to be futile.

A more suitable candidate for parallelization is the Swendsen-Wang cluster algorithm. Although we were able to create an implementation using OpenMP, porting it to CUDA was without any success. Due to the advantages of the single-cluster algorithm over the Swendsen-Wang algorithm, we decided to create a trivial parallelized implementation of the single-cluster algorithm for the CPU using OpenMP. Nevertheless, we want to give some remarks on implementing the Swendsen-Wang algorithm in parallel, and also some remarks on what was our problem with porting it to CUDA:

- On the part of the CPU, a simple approach is to divide the lattice into sublattices and to apply one of the commonly known cluster identification algorithms, such as 'ants in the labyrinth' or the Hoshen-Kopelman algorithm, within each of the sublattices. This can be done in parallel. Having finished this task, all clusters within the sublattices are correctly labeled, except for those that extend across more than one sublattice. These clusters require additional work. Unfortunately, this part is bad to parallelize, and thus should be done by some kind of master thread in order to avoid race conditions in resolving the cluster labels.

  An implementation for the CPU that uses the 'cluster-self-labeling' method,[5] proposed by Coddington and Baillie [6], was created in the context of this thesis. Figure 5.10 shows how this implementation scales on an Intel Core i7-920.

- The problem with porting our CPU implementation to CUDA results from the existence of a master thread which resolves labels from clusters that extend across more than one sublattice. Matching the final cluster labels by using just one CUDA thread leads to a very poor workload of the graphics card. To get rid of this master

[5] The idea is to make every site to look in turn at each of its bounded neighbors and to set the involved labels each to the minimum of the two. This procedure iterates until nothing changes.

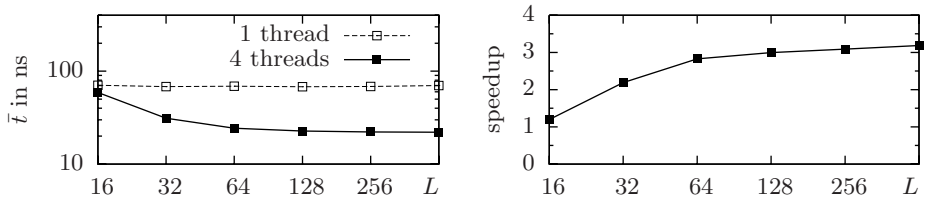

Figure 5.10.: Mean update times $\bar{t}$ per spin, and speedups for simulating the two-dimensional Ising model on squared lattices with periodic boundary conditions and extents L. We used a parallelized CPU implementation of the Swendsen-Wang algorithm.

thread, we optimized the matching of the final cluster labels by means of atomic functions which ensure thread-exclusive memory operations, necessary to avoid race conditions in resolving the cluster labels in parallel. Unfortunately, extensive use of atomic functions made our code rather slow.

As far as we can tell, non-local update algorithms, such as cluster algorithms, suffer from the circumstance that CUDA threads that belong to different thread blocks cannot synchronize their activities to each other, which hence results in the necessity of mechanisms that compensate for this lack. Since appropriate approaches contradict somehow with the CUDA programming model, processing cluster algorithms on current graphics cards seems less promising. Similar investigations were done by [24].[6] The author's implementation of the Swendsen-Wang algorithm using CUDA gives update times per spin of 29ns when running one cluster simulation on the GPU (our implementation for the CPU gives update times per spin of down to 22ns). Although the author reports on difficulties (similar to ours) in porting the Swendsen-Wang algorithm to CUDA, he was able to lower mean update times per spin to 2.9ns by running several cluster simulations in parallel on the GPU. Compared to our CPU implementation, [24] thus has shown that speedups of about a factor 7.5 can be achieved when using the Tesla C1060 instead of an Intel Core i7-920 quad-core CPU. In a certain sense, this value agrees with our speedups from Section 5.1.3.

As mentioned, we created a trivial parallelized implementation of the single-cluster algorithm for the CPU in order to get rid of critical slowing down, and at the same time to make use of the CPU's capabilities. On an Intel Core i7-920, we measured effective update times per spin of about 6.6ns,[7] which is of the same order as [24]'s 2.9ns for the Swendsen-Wang algorithm using CUDA. Since the single-cluster algorithm's dynamical exponent is smaller than that of the Swendsen-Wang algorithm, and also because the class of spin models for which efficient cluster algorithms exist is really small, we stopped our effort of porting the Swendsen-Wang algorithm to CUDA.

[6] We came across this work about 1 month before we finished this thesis.

[7] The calculation of the update times per spin involve the mean cluster size which considers the number of actually flipped spins.

5.2.2. Single-cluster algorithm—On testing parallel random numbers

As already noted, the single-cluster algorithm is extremely sensitive to subtle correlations within random number sequences. Following the argumentation in [10], sequences of random numbers with high order bits equal to zero will not be random in these bits, that is, these bits may remain zero in newly generated random numbers. Since the single-cluster algorithm compares random numbers with only a single bond probability, this effect may lead to a very small bias in the size of the clusters that are generated. As a consequence, systematic deviations of Monte Carlo estimates of system quantities may occur.

In turn, this property allows to test our parallel random number generators for their quality, for instance, by comparing Monte Carlo estimates of the internal energy and the specific heat of the two-dimensional Ising model with exact calculations. For this

	pranlxs|0	pranlxd|1	pmts	pmtd	CDAran32	CDAran64
	1.453066(40)	1.453132(40)	1.453065(40)	1.453158(40)	1.453075(40)	1.453046(40)
	1.453076(40)	1.453057(40)	1.453095(40)	1.453039(40)	1.453020(40)	1.453033(40)
	1.453113(40)	1.453011(40)	1.453060(40)	1.453065(40)	1.453161(40)	1.453085(40)
	1.453047(40)	1.453041(40)	1.453042(40)	1.453006(40)	1.453092(40)	1.453052(40)
	1.453066(40)	1.453033(40)	1.453071(40)	1.453089(40)	1.453072(40)	1.453063(40)
	1.453035(40)	1.453080(40)	1.453046(40)	1.453024(40)	1.453067(40)	1.453135(40)
	1.453088(40)	1.453073(40)	1.453113(40)	1.453057(40)	1.453003(40)	1.453039(40)
	1.453100(40)	1.453078(40)	1.453054(40)	1.453065(40)	1.453077(40)	1.453013(40)
	1.453097(40)	1.453062(40)	1.453050(40)	1.452957(40)	1.453019(40)	1.453091(40)
	1.453002(40)	1.453073(40)	1.453028(40)	1.453035(40)	1.453065(40)	1.453112(40)
$-E$	1.453069	1.453064	1.453062	1.453050	1.453065	1.453067
error	0.000013	0.000013	0.000013	0.000013	0.000013	0.000013
dev.	0.319σ	-0.065σ	-0.219σ	-1.142σ	0.012σ	0.165σ
	1.49937(40)	1.49856(40)	1.49871(40)	1.49786(40)	1.49823(40)	1.49865(40)
	1.49920(40)	1.49819(40)	1.49825(40)	1.49882(40)	1.49916(40)	1.49903(40)
	1.49822(40)	1.49897(40)	1.49839(40)	1.49868(40)	1.49801(40)	1.49887(40)
	1.49912(40)	1.49842(40)	1.49913(40)	1.49907(40)	1.49909(40)	1.49858(40)
	1.49856(40)	1.49781(40)	1.49831(40)	1.49882(40)	1.49829(40)	1.49886(40)
	1.49931(40)	1.49872(40)	1.49864(40)	1.49845(40)	1.49819(40)	1.49858(40)
	1.49842(40)	1.49811(40)	1.49825(40)	1.49872(40)	1.49882(40)	1.49894(40)
	1.49843(40)	1.49849(40)	1.49847(40)	1.49805(40)	1.49849(40)	1.49896(40)
	1.49900(40)	1.49928(40)	1.49877(40)	1.49888(40)	1.49864(40)	1.49845(40)
	1.49900(40)	1.49883(40)	1.49849(40)	1.49872(40)	1.49884(40)	1.49884(40)
c_V	1.49886	1.49854	1.49854	1.49860	1.49858	1.49878
error	0.00013	0.00013	0.00013	0.00013	0.00013	0.00013
dev.	1.193σ	-1.269σ	-1.269σ	-0.807σ	-0.961σ	0.577σ

Table 5.2.: Monte Carlo estimates of the internal energy per spin (the listed quantity is $-E$) and the specific heat per spin from Monte Carlo simulations of the two-dimensional Ising model on a (16×16) squared lattice using the single-cluster algorithm. Exact calculations [9]: $E = -1.45306485\ldots$ and $c_V = 1.49870495\ldots$

purpose we simulated the two-dimensional Ising model on a (16×16) and a (32×32) squared lattice by means of the single-cluster algorithm with random numbers from generators used in this thesis. For each generator we produced a total of 10 replica, each containing $100000000 = 10^8$ estimates of the internal energy and the specific heat. The statistical errors of these quantities were calculated with the `UWerr` script, where, due to the large amount of measurement data, all replica were analyzed separately. Since they are assumed to be independent from each other, the total error can be calculated as weighted average over these partial issues. Simulation results are listed in Table 5.2 and Table E.6 in Appendix E.4. We used the abbreviations listed in Table 5.1.

All random number generators give estimates of the internal energy and the specific heat which agree with exact calculations within errors. We thus conclude that all random number sequences used for the simulations do not show any discrepancies. Retrospectively, this in particular justifies the use of the random number generators CDAran32 and CDAran64 for parallel Monte Carlo simulations of Ising-like spin models. The same is also true for the other random number generators used in this thesis.

5.2.3. Single-cluster algorithm—Critical exponents of the Ising model

Most simulations of statistical systems at criticality aim for the determination of critical exponents. One possibility to obtain critical exponents of the Ising model is to measure system quantities for temperatures T close to the critical temperature, and to extract the exponents from the corresponding power laws, given in Section 1.4.1. Since these power laws are valid only in the thermodynamic limit, this approach requires the system to be simulated on lattices with large extents L. Even if we would use the single-cluster algorithm to perform these simulations, it would take us a huge amount of time to simulate a sufficient large set of Ising systems with temperatures in the vicinity of T_c. Further difficulties result from ambiguities about what is actually meant with 'in the vicinity of T_c'.

A more sophisticated approach to obtain critical exponents is to use finite size scaling methods, as detailed in Section 1.4.2. Estimating, for instance, the magnetic susceptibility

$$\chi \propto L^{2-\eta}\Big(1 + A_1 t L^{1/\nu} + A_2 u_3^0 L^{-\omega} + \dots\Big) \tag{5.6}$$

for lattices with different extents L then allows to deduce the critical exponents η, ν and $\omega = y_3$. The remaining exponents follow from scaling relations.

The exponent ν can be also deduced from the *fourth-order Binder cumulant*

$$C_4 = 1 - \frac{\langle m^4 \rangle}{3\langle m^2 \rangle^2}\,. \tag{5.7}$$

As T approaches T_c, C_4 becomes a universal constant. This property allows to estimate the critical point itself, for instance, by plotting C_4 for a set of different lattice extents. Due to C_4's universality at criticality, the corresponding graphs are expected to intersect at the critical point. Subsequently, the focus will be on the dimensionless ratio $Q =$

$\langle m^2\rangle^2/\langle m^4\rangle$. Near the critical point, Q behaves as [8]

$$Q(t, u_3^0, L) = Q\Big(tL^{1/\nu}, u_3^0 L^{-\omega}, 1\Big) + \dots\,, \tag{5.8}$$

where t is the thermal scaling variable and u_3^0 is some irrelevant scaling variable. Taylor expansion of the right hand side of Eq. (5.8) yields

$$\begin{aligned} Q(t, u_3^0, L) = {} & Q^{(0,0)} + Q^{(1,0)} t L^{1/\nu} + Q^{(2,0)} t^2 L^{2/\nu} + \\ & + Q^{(0,1)} u_3^0 L^{-\omega} + Q^{(1,1)} t u_3^0 L^{-\omega/\nu} + \dots\,, \end{aligned} \tag{5.9}$$

where derivatives of the universal function Q with respect to t and/or u_3^0 are denoted as $Q^{(i,j)}$. Differentiating Eq. (5.9) with respect to t then gives

$$\frac{\partial Q}{\partial t} \propto L^{1/\nu}\Big(1 + A_1 t L^{1/\nu} + A_2 u_3^0 L^{-\omega} + \dots\Big)\,. \tag{5.10}$$

Since $t = t(\beta)$, the computational task will be on estimating the derivative of the universal function Q with respect to β. To do so, we need the following relation:

$$\begin{aligned} \frac{\partial}{\partial\beta}\langle A\rangle &= \frac{\partial}{\partial\beta}\left(\frac{1}{Z}\sum_i A_i\,\mathrm{e}^{-\beta E_i}\right) \\ &= \left(-\frac{1}{Z^2}\frac{\partial}{\partial\beta}\sum_i \mathrm{e}^{-\beta E_i}\right)\sum_j A_j\,\mathrm{e}^{-\beta E_j} + \frac{1}{Z}\left(\frac{\partial}{\partial\beta}\sum_i A_i\,\mathrm{e}^{-\beta E_i}\right) \\ &= -\frac{1}{Z^2}\left(\sum_i (-E_i)\,\mathrm{e}^{-\beta E_i}\right)\sum_j A_j\,\mathrm{e}^{-\beta E_j} + \frac{1}{Z}\left(\sum_i A_i(-E_i)\,\mathrm{e}^{-\beta E_i}\right) \\ &= \langle E\rangle\langle A\rangle - \langle AE\rangle\,. \end{aligned} \tag{5.11}$$

Thus,

$$\begin{aligned} \frac{\partial Q}{\partial\beta} = \frac{\partial}{\partial\beta}\frac{\langle m^2\rangle^2}{\langle m^4\rangle} &= 2\,\frac{\langle m^2\rangle}{\langle m^4\rangle}\frac{\partial}{\partial\beta}\langle m^2\rangle - \frac{\langle m^2\rangle^2}{\langle m^4\rangle^2}\frac{\partial}{\partial\beta}\langle m^4\rangle \\ &= \frac{\langle m^2\rangle^2\langle H\rangle}{\langle m^4\rangle} - 2\,\frac{\langle m^2\rangle\langle m^2 H\rangle}{\langle m^4\rangle} + \frac{\langle m^2\rangle^2\langle m^4 H\rangle}{\langle m^4\rangle^2}\,. \end{aligned} \tag{5.12}$$

For both the two-dimensional Ising model and the three-dimensional Ising model, we evaluated the quantities χ, C_4 and $\partial Q/\partial\beta$ for different extents of the lattice. We used the single-cluster algorithm and the Ranlux random number generator pranlxd|1.

While for the two-dimensional Ising model the critical point is exactly known, for the three-dimensional Ising model only estimates of it are available. Unfortunately, we performed all simulations of the three-dimensional Ising model at inverse temperature $\beta_c^* = 0.22165$ [2]. At present there are more precise estimates of β_c [8]: $\beta_c =$

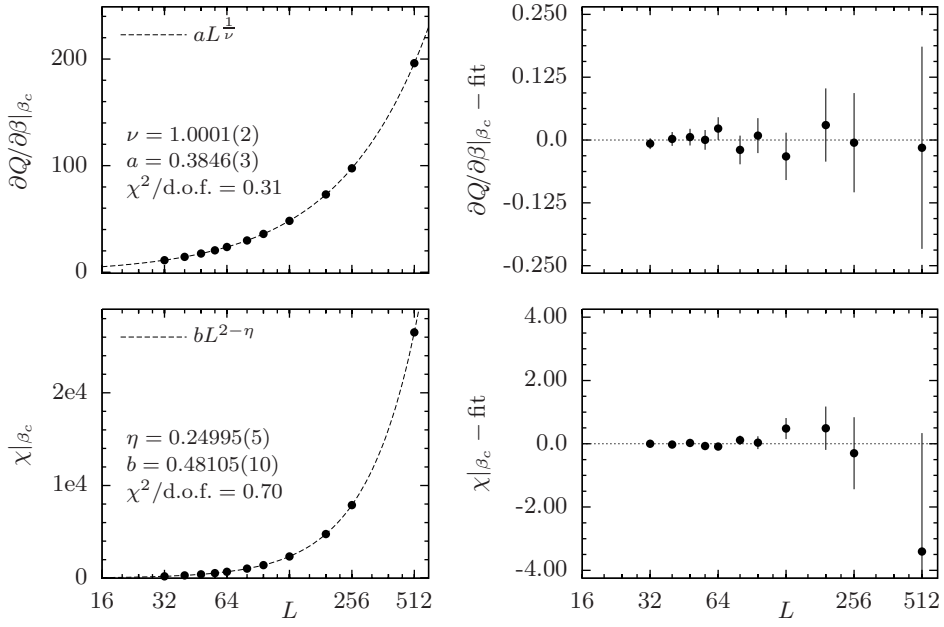

Figure 5.11.: Least squares fits to estimates of χ and $\partial Q/\partial\beta$ from simulating the two-dimensional Ising model by means of the single-cluster algorithm.

0.22165455(3). We used the single histogram method (see Appendix E.5.1) to extrapolate system quantities from $\beta_c^* = 0.22165$ to $\beta_c = 0.22165455$. Estimates from simulating the two-dimensional and the three-dimensional Ising model are listed in Tables E.7 and E.8 in Appendix E.5. The critical exponents ν, η and ω then follow from adjusting Eqs. (5.6) and (5.10) to the (extrapolated) estimates of χ and $\partial Q/\partial\beta$.

For the two-dimensional Ising model, ω is known to be 2 for squared lattices [20], so that terms proportional to $L^{-\omega}$ give negligible corrections to the leading order terms in Eqs. (5.6) and (5.10). Since all simulations were performed directly at the critical point, t is equal to zero. Neglecting all corrections to the leading order terms, Eqs. (5.6) and (5.10) then lead to

$$\chi \propto L^{2-\eta} \qquad \text{and} \qquad \frac{\partial Q}{\partial \beta} \propto L^{\frac{1}{\nu}} \,. \tag{5.13}$$

Figure 5.11 shows the results of least squares fits[8] to the estimates given in Table E.7 in Appendix E.5. The estimates $\nu = 1.0001(2)$ and $\eta = 0.24995(5)$ are in good agreement with the exact values $\nu_{\text{exact}} = 1$ and $\eta_{\text{exact}} = 0.25$. They lead to the critical exponents

$$\gamma = 1.7502(4)\,, \qquad \alpha = -0.0002(4)\,, \qquad \beta = 0.1250(4)\,, \qquad \delta = 15.00(5)\,,$$

which are also in good agreement with the exact values.

For the three-dimensional Ising model, ω is known to be about 0.8 for cubic lattices [20]. In the case of small lattices, terms that are proportional to $L^{-\omega}$ thus give non-

[8] We used the program Gnuplot to perform least squares fits. `http://www.gnuplot.info/`

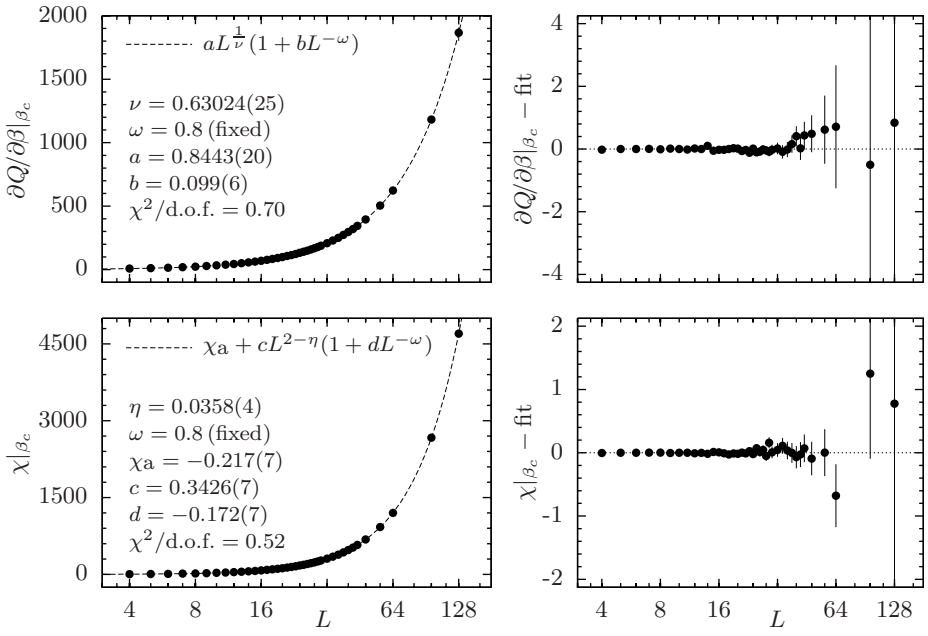

Figure 5.12.: Least squares fits to estimates of χ and $\partial Q/\partial\beta$ from simulating the three-dimensional Ising model by means of the single-cluster algorithm. The parameter ω was set to $\omega = 0.8$ fixed.

negligible corrections to the leading order terms in (5.6) and (5.10), other than in the two-dimensional case. Using Eqs. (5.13) to fit our data, we then are to consider only data points that correspond to large lattice extents. Fitting the estimates from Table E.8 with these models leads to values $\chi^2/\text{d.o.f.} \gg 1$, which makes them inappropriate to describe our data.

Considering corrections to the leading order terms, and assuming that $\beta_c = 0.22165455$ is extremely close to the critical point, t can be set to zero, and estimates from Table E.8 should be well represented by

$$\chi \propto L^{2-\eta}\left(1 + C_1 L^{-\omega}\right) \quad \text{and} \quad \frac{\partial Q}{\partial \beta} \propto L^{\frac{1}{\nu}}\left(1 + C_2 L^{-\omega}\right). \tag{5.14}$$

Since the parameter ω is hard to estimate, we considered literature values [8]. For fixed $\omega \in [0.75, 0.85]$, the remaining parameters were adjusted to the extrapolated data from Table E.8. To obtain good agreement in fitting the magnetic susceptibility χ, we introduced an additional constant χ_a which corresponds to the analytic part of the free energy density—in fitting the quantity $\partial Q/\partial\beta$, such a constant was not necessary—. Table E.9 in Appendix E.5 summarizes the results of least squares fits for different ω values.

Varying ω leads to small changes of the fit-parameters, but allmost all of them are compatible within errors. Also, all ω values between 0.75 and 0.85 yield critical exponents that are in agreement with literature values [8],[20]. Figure 5.12 shows the results of least squares fits with $\omega = 0.8$ fixed.

6. Simulating the Ising spin glass on parallel computer architectures

This chapter is concerned with the Ising spin glass (ISG) as a particular instance of the general class of models known as Edwards-Anderson spin glasses. Since simulating the ISG by means of Monte Carlo methods is very time-consuming, especially when aiming to be comparable with literature values, we will focus on investigating execution times for simulating the ISG on Tesla C1060 graphics cards as well as on the Intel Core i7-920 quad-core CPU.

To introduce into the subject of simulating the ISG [15], we first consider the phenomenon of *ergodicity breaking* as a consequence of the existence of many non-symmetry-equivalent ground states, as is the case for spin glasses. To get rid of ergodicity breaking within our Monte Carlo simulations, we then consider the *parallel tempering method*, commonly used to simulate spin glasses on parallel computers. Finally we will present execution times for updating spin configurations of the ISG.

6.1. Ergodicity breaking and the parallel tempering method

As figured out in Section 1.3, the presence of both ferromagnetic and anti-ferromagnetic interactions leads to frustrated spin systems with a large set of possible ground states, separated by high energy barriers. If we want to simulate such a system in its glassy phase (i.e. $T < T_g$) by means of the Metropolis algorithm our simulation would stuck within a small region of the phase space. The point is that the Metropolis algorithm samples phase states according to their Boltzmann weights, which results in rejecting almost all spin configurations that would significantly increase the system's energy. Using the Metropolis algorithm, our simulation thus would not be able to make the system cross high energy barriers, that is, it strands in one of the energy basins.[1] As a consequence, ergodicity is broken.

Since ergodicity breaking reflects that the real system also has trouble dealing with escaping the energy basins it stranded in, it is not a failure of the Metropolis algorithm to stuck within these basins in our simulation. Nevertheless, investigating the ISG in its glassy phase by means of the Metropolis algorithm becomes a problem as system properties will depend on the particular energy basin the system is stranded in.[2]

[1] Which energy basin the system moves to, strongly depends on the realization of the randomness of the interactions, and the random number sequences used for the updates.

[2] As mentioned in Section 1.2, the Hamiltonian of the Ising model in zero magnetic field also exhibits more than one ground state—2 to be precise—, but both of them are symmetry-equivalent. It therefore does not matter which ground state we are into when taking measurements.

There are two common ways to overcome ergodicity breaking [15]. The first is to use an algorithm that samples high-energy states with greater than the appropriate Boltzmann weight, making the system more likely to cross energy barriers. The other alternative, known as *simulated tempering method*, is to generate states in a way which allows the system to jump to very different configurations, and so to cross energy barriers without passing through a whole set of intermediate states. For simulating spin glasses on parallel computer architectures, the *parallel tempering method*, a variation of the simulated tempering method, is commonly used.

The idea behind the parallel tempering method, as detailed in Appendix F.1, is to perform several simulations of glassy systems in parallel, each of them with the same realization of the spin-spin interactions, but at different temperatures. Updating the spin configurations of these systems can be done, for instance, by means of the Metropolis algorithm, which in the case of the ISG works the same as for the Ising model. From time to time one interchanges the spin configurations between two of the simulations (swap move) with a certain probability which ensures that the states within each simulation still follow the correct Boltzmann distribution at the appropriate temperature. In this way higher-temperature simulations (commonly at temperatures $T \gtrapprox T_g$) help lower-temperature simulations ($T < T_g$) to cross energy barriers.

Implementing the parallel tempering method on parallel computer architectures is straightforward, since all methods/techniques described in the previous chapter, such as the checkerboard procedure, are also applicable to the ISG—this was meant in Chapter 5 when having said 'An implementation that allows to simulate the Ising model, also allows to simulate any other Ising-like spin model by just doing simple modifications of the source code'—. To investigate execution times for simulating the ISG on Tesla C1060 graphics cards and on the Intel Core i7-920, it thus should be sufficient to consider the Metropolis update within any of the systems used by the parallel tempering method. We therefore created implementations of the checkerboard procedure for the two- and the three-dimensional bimodal bond distributed ISG as well as for the two- and the three-dimensional Gaussian bond distributed ISG. Our implementations for the GPU use CUDA, and those for the CPU use OpenMP. All implementations are designed to perform calculations with single-precision as well as with double-precision. Listing F.1 in Appendix F.2 shows the source code of our CUDA kernel that realizes the first step of the checkerboard procedure for the two-dimensional bimodal bond distributed ISG, according to Figure 5.1.

6.2. Execution times

In this section we present mean update times per spin for simulating the ISG on Nvidia Tesla C1060 graphics cards and on an Intel Core i7-920 quad-core processor. Similar to the runtime measurements in the case of the Ising model, the number of Monte Carlo sweeps n that separate consecutive measurements was set to $n = 20$, and the total number of Monte Carlo sweeps N was chosen to be much larger than n.

As can be seen from Figures 6.1-6.4, mean update times per spin are slightly larger

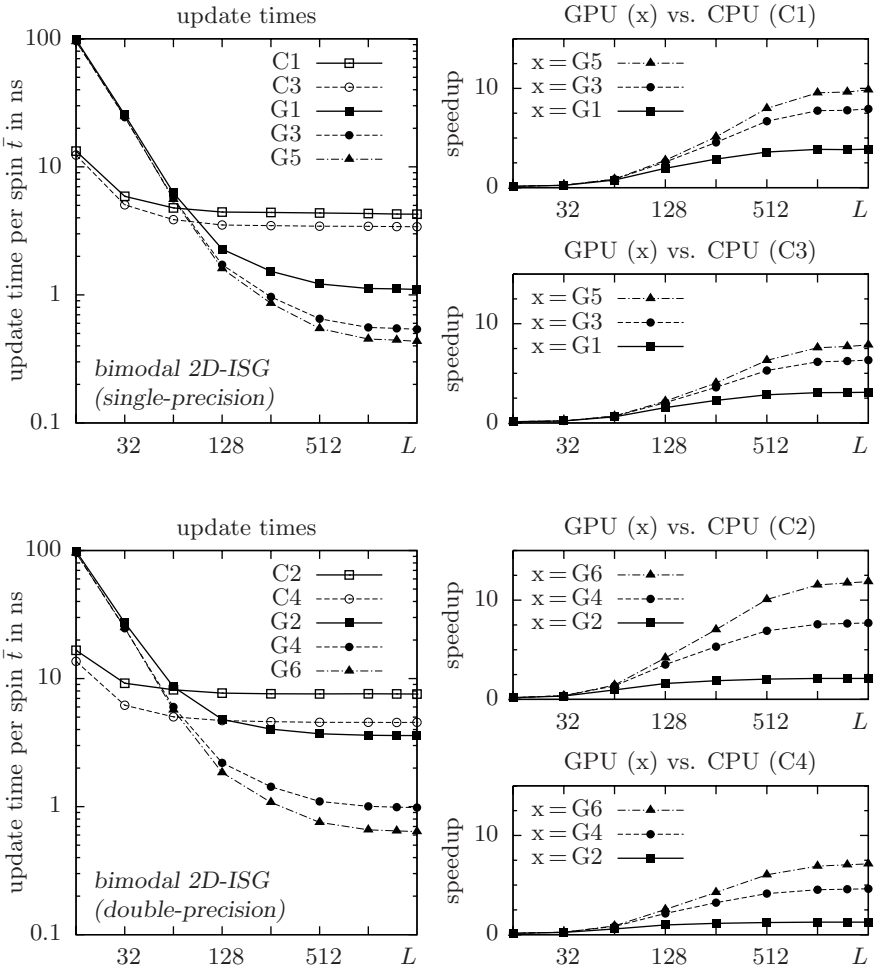

Figure 6.1.: Mean update times per spin $\bar{t}$, and speedups for simulating the two-dimensional bimodal bond distributed ISG on squared lattices with periodic boundary conditions and extents L. Abbreviations for random number generators are given in Table 5.1. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

than those for the Ising model (see Figures 5.5 and 5.6). Other than the Ising model, the ISG involves both ferromagnetic and anti-ferromagnetic spin-spin interactions J_{ij}, stored in the main memory of the respective device. Since almost all calculations concerning the update of the spins involve the couplings J_{ij}, the number of memory requests becomes much larger than in the case of simulating the Ising model. In particular, the couplings J_{ij} appear in the nearest-neighbor sums $x = \sum_{\langle ij \rangle} J_{ij} s_i s_j$, which results in an increase in the computational effort in order to perform the Metropolis update steps. In the case of the Gaussian bond distributed ISG, updating spins also requires to evaluate many exponential functions—due to $J_{ij} \notin \mathbb{Z}$, and hence $x \notin \mathbb{Z}$, the number of acceptance ratios $A(\mu \to \nu)$ is too large to efficiently use precomputed values of them—.

We therefore expect the Ising model to display the lowest update times per spin, and

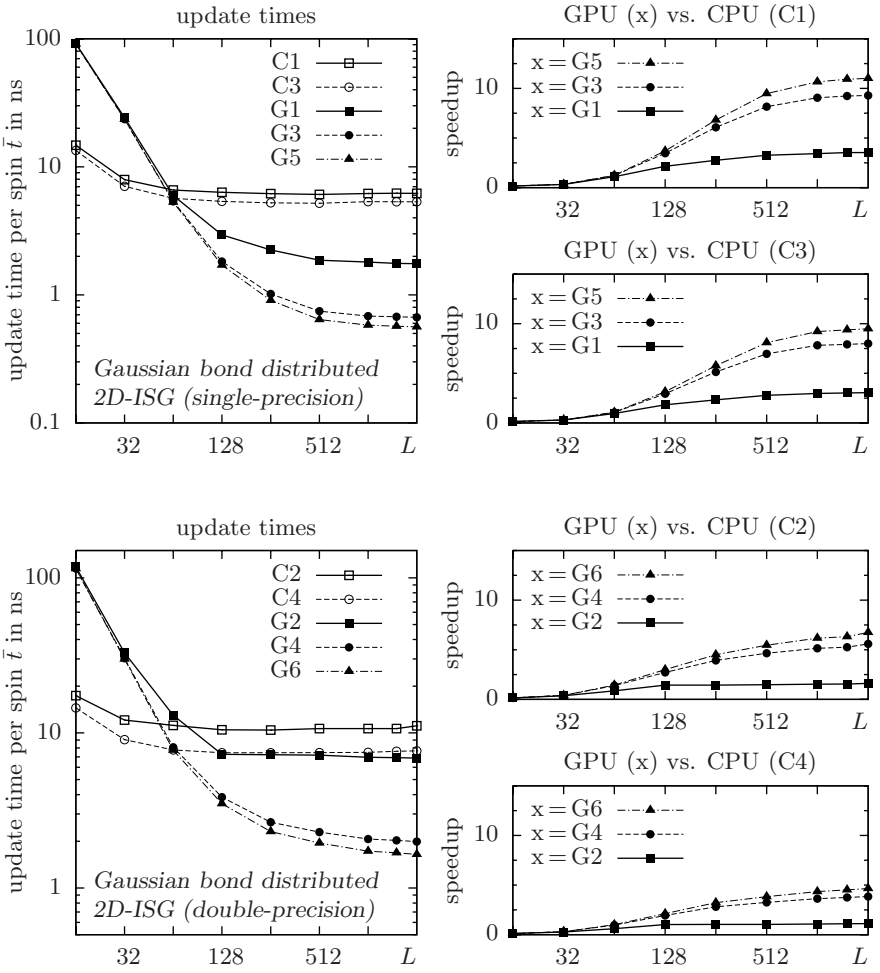

Figure 6.2.: Mean update times per spin $\bar{t}$, and speedups for simulating the two-dimensional Gaussian bond distributed ISG on squared lattices with periodic boundary conditions and extents L. Abbreviations for random number generators are given in Table 5.1. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

the Gaussian bond distributed ISG to display the largest update times per spin. In fact this is what was actually observed. Figures 6.5-6.6 contrast update times per spin for simulating different spin models on squared/cubic lattices with periodic boundary conditions and extent $L = 2048$ in two dimensions and $L = 256$ in three dimensions.

As was to be expected, restricting all calculations to be performed with single-precision, both the Tesla C1060 and the Intel Core i7-920 show a similar behavior of the increase in the mean update times per spin when switching from the Ising model to the bimodal bond distributed ISG and then to the Gaussian bond distributed ISG. Since there are no reasons why the Tesla C1060 should be able to perform single-precision calculations much faster than the Intel Core i7-920, or vice versa, this outcome is not surprising.

Doing all calculations with double-precision, both the Tesla C1060 and the Intel Core

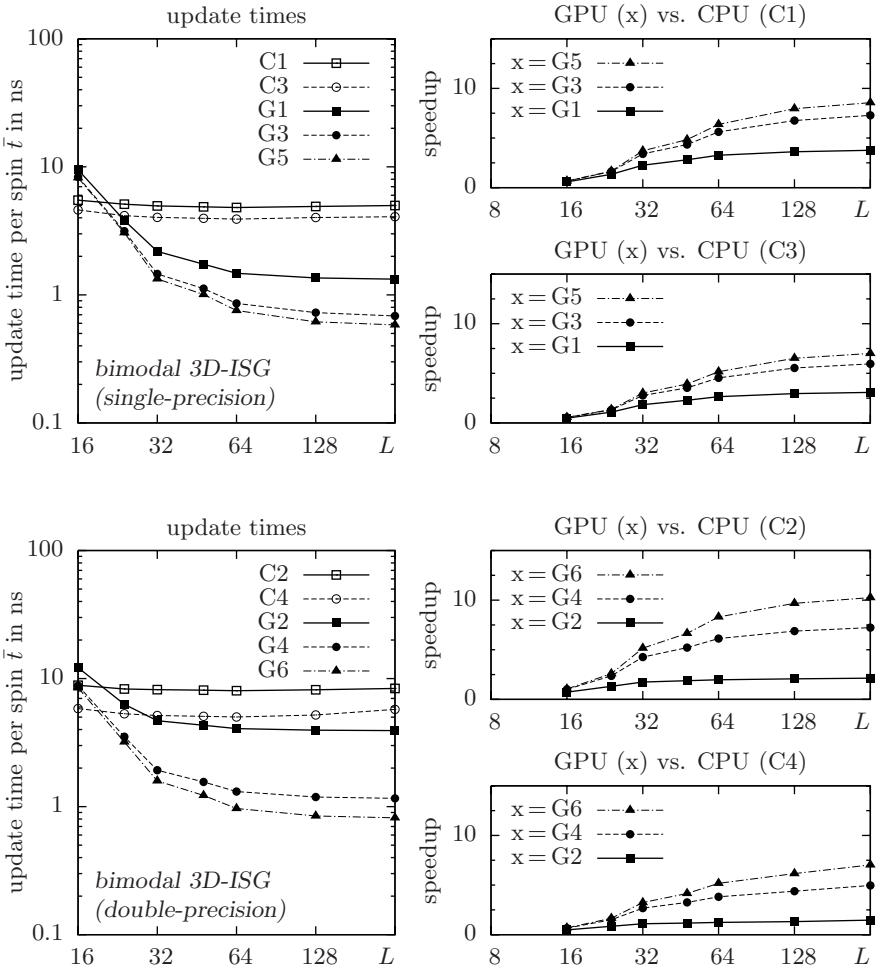

Figure 6.3.: Mean update times per spin $\bar{t}$, and speedups for simulating the three-dimensional bimodal bond distributed ISG on cubic lattices with periodic boundary conditions and extents L. Abbreviations for random number generators are given in Table 5.1. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

i7-920 also show a similar behavior of the increase in the mean update times per spin, except for simulating the Gaussian bond distributed ISG. Since the latter model makes both the Tesla C1060 and the Intel Core i7-920 perform almost only double-precision calculations, the GPU suffers from the circumstance that 8 scalar processors share one double-precision unit per multiprocessor, whereas the Core i7-920 performs double-precision calculations with about half of its single-precision performance, as expected from current CPUs. Nevertheless, the Tesla C1060 is able to simulate the Gaussian bond distributed ISG about 5 times faster than the Intel Core i7-920, even if all calculations are performed with double-precision.

Summarizingly, speedups for simulating the ISG on graphics cards instead of CPUs are almost the same as those from the previous chapter. We measured maximum speedups

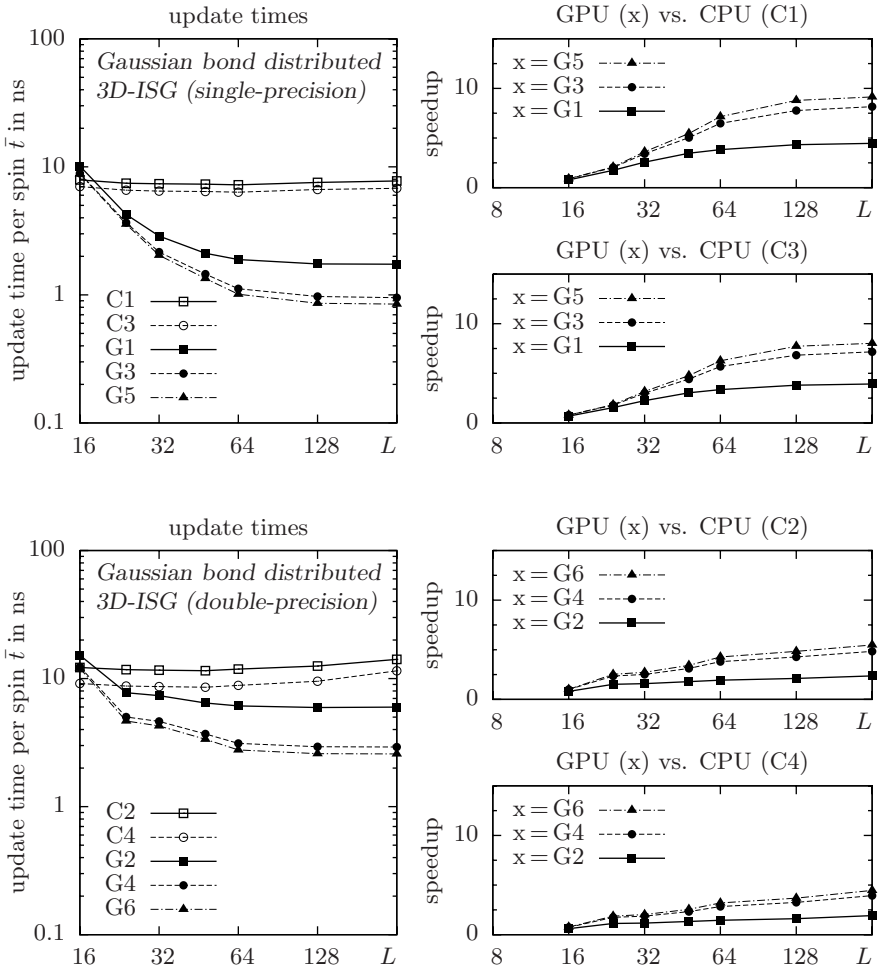

Figure 6.4.: Mean update times per spin $\bar{t}$, and speedups for simulating the three-dimensional Gaussian bond distributed ISG on cubic lattices with periodic boundary conditions and extents L. Abbreviations for random number generators are given in Table 5.1. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

which are about 10 for calculations with single-precision, whereas maximum speedups for calculations with double-precision are about 5.

Although there are some possibilities to speedup our codes, such as multi-spin-coding[3] or double-checkerboard update (see Section 5.1.4), we want to note that all improvements

[3]The idea behind multi-spin-coding is to store states of several spin variables in the bits of a computer word, which in the case of a 32-bit machine allows to have spins from 32 lattice sites within a single integer variable. Commonly, these 32 spins correspond to the same point on the lattice but belong to independent simulations (asynchronous multi-spin-coding), which then are processed in parallel. Since multi-spin-coding involves low-level bit manipulations, algorithms need to be adjusted in a rather special way, where only those that are less complex benefit from this technique. On a related note, multi-spin-coding also requires one to switch from standard random number generators to those that allow to produce random bits with a given probability distribution.

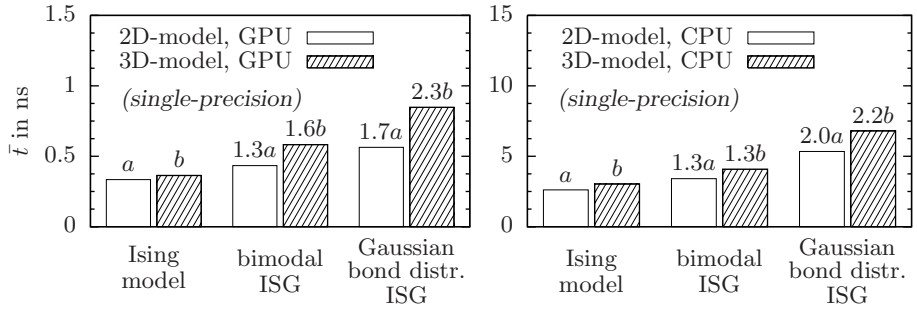

Figure 6.5.: Mean update times $\bar{t}$ per spin for simulating two- and three-dimensional spin models (single-precision) on squared lattices with extent $L = 2048$, and cubic lattices with extent $L = 256$. The numbers sitting on top of the bars reflect the increase in the mean update times per spin, compared those in the case of the Ising model. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

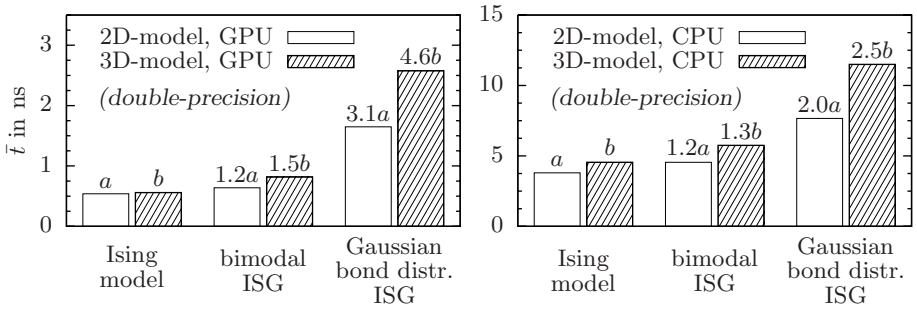

Figure 6.6.: Mean update times $\bar{t}$ per spin for simulating two- and three-dimensional spin models (double-precision) on squared lattices with extent $L = 2048$, and cubic lattices with extent $L = 256$. The numbers sitting on top of the bars reflect the increase in the mean update times per spin, compared those in the case of the Ising model. For all runtime measurements, we used Nvidia Tesla C1060 graphics cards and an Intel Core i7-920 quad-core CPU.

that will be done for the GPU also have to be done for the CPU. Since there are no reasons why the GPU should benefit from them in a more special way than the CPU, only the total runtimes will decrease, but speedups will not change significantly. Since we are only interested in comparing the performance of the GPU with that of the CPU, we have not implemented such improvements—speedups will be unaffected—.

7. Discussion

After having investigated different approaches to map the CUDA programming model onto spin systems, we now want to give some retrospective remarks on what was done. In particular this chapter concerns aspects on how to reliably compare the performance of the GPU with that of the CPU, as well as the programming effort for writing CUDA code and parallelized CPU code respectively. In a further section we consider the reliability of GPGPU computations, especially with respect to overheating of the graphics cards and the occurrence of 'thermal induced soft-errors'. Also some information about Nvidia's next generation GPUs will be given.

7.1. On reliably comparing the performance of GPU and CPU

As already mentioned a few times before this chapter, there is a lot of confusion about how to reliably compare the performance of the GPU against that of the CPU. As shown by strongly varying outcomes of papers concerning this subject views are very different. While `http://www.nvidia.com/object/cuda_showcase_html.html` cites papers that report about astronomically large performance improvements by using CUDA-capable GPUs instead of standard CPUs, our present work gives less optimistic results. Of course, there are scientific issues that benefit from the graphics hardware in a (much) more special way than others, but in principal almost all speedups that attest the GPU to absolutely outperform current multi-core CPUs suffer extremely bad CPU implementations [21, 7]. On the other hand, the circumstance that many people are able to write fast graphics card programs certifies CUDA to be a simple-to-use approach for massively parallel programming. In fact, the introduction of OpenCL[1] (Open Computing Language) in 2009/2010—significantly influenced by the CUDA programming model, and also developed in cooperation with Nvidia—supports this programming model to possibly become the future-framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors.

Now returning to the subject, there is still the question about how to compare the GPU against the CPU. As always noted throughout this thesis, we made use of OpenMP to make our implementations perform on Intel quad-core CPUs, Core i7-920 and Xeon E5520 namely. In almost all cases we measured performance improvements of about a factor $3 - 4$, compared to non-multi-threaded CPU implementations. A crucial reason for these improvements follows from the circumstance that current CPUs feature large (shared) data caches, which for almost all investigated models allow to have even large lattices within the cache.

[1] `http://www.khronos.org/`

In addition to OpenMP, we also optimized our CPU codes for SSE by means of SSE intrinsics, inline assembly or simply rearranging compute-intensive code segments in order to allow the compiler to optimize for SSE. On average, we obtained runtime improvements of about a factor $1.5 - 2.5$ against code that is not optimized for SSE.

In our opinion, a fair comparison between CPU and GPU requires to incorporate at least these two aspects when writing CPU code. In so doing, performance improvements of about a factor $5 - 10$ can be achieved, compared to non-optimized CPU code. Further gains strongly depend on the compiler used. Throughout this thesis, we determined the GNU compiler `g++` [2] to do its job almost always as good as the Intel compiler `icpc`.[3] Nonetheless, it might be important to consider appropriate compiler flags—choosing optimization level `-O3` on the part of the GNU compiler is a good idea, but additional compiler flags, for instance as given in Footnote 2, may significantly gain the performance of an application. The same also applies for the Intel compiler (see Footnote 3)—.

To summarize what was said, the following performance issues for the CPU need to be considered in order to get reliable benchmark results:

- make use of multi-threading APIs/libraries such as OpenMP or Pthreads,
- optimize the code for SSE or make the compiler do this job,
- choose appropriate compiler (optimization) flags to achieve best performance.

On the part of the GPU, similar considerations also apply, but other than on the CPU, the CUDA programming model involves multi-threading anyway, that is, there is no need for external libraries.

In addition to these rather technical aspects, one needs to consider what is actually compared against each other. This points aims, for instance, at the double-checkerboard procedure. At first sight, this method indeed acts the same as the checkerboard procedure, but it seems to perform much faster. As detailed in Section 5.1.4, the double-checkerboard procedure modifies the original implementation in such a way that autocorrelation times significantly increase, concurrently to the decrease in execution times, both compared to simulations that use the checkerboard procedure. As a consequence, the overall performance only slightly increases. Performance improvements due to using the double-checkerboard procedure instead of the checkerboard procedure therefore requires to incorporate changes in the autocorrelation times.

A somewhat different example is given by the implementation of the Ranlux random number generator presented in [7]. The author seems to produce single random numbers, each of them as result of an independent function call. Producing random numbers this way on the CPU, may result in an extremely large function-call overhead.[4] On the

[2] We used `g++` version 4.3.2 with compiler flags: `-O3 -msse4.2 -mfpmath=sse,387 -fopenmp`.

[3] We used `icpc` version 11.1, compiler flags: `-fast -axSSE4.2 -openmp -static-intel`.

[4] In the case of the Ranlux random number generator we assume the Microsoft compiler, as used by the author, to not inline Ranlux's update function, even if declared as `inline`. The reason is that Ranlux is a rather complex random number generator, and its update procedure leads to large codes. The compiler therefore is free to decide whether to inline it or not—most likely it rejects inlining—. The only way to enforce the Microsoft compiler to do this anyhow, is to make use of appropriate compiler flags or to use `__forceinline` for the update function. Since the author does not provides information about this point, we assume all Ranlux calls to lead to an extremely large function-call overhead.

part of the GPU, the thread scheduler should be able to hide this function-call overhead by processing independent threads while other threads are waiting.[5] In essence, the CPU implementation is per se at a disadvantage, and in fact this is what the author's benchmark results reflect.

A strange observation in this respect is that many authors are not clear about these points, or they consciously ignore them. In addition to that, the performance of up-to-date high-end graphics devices is conspicuously often compared against the performance of early quad-core CPUs. One of the standard benchmark setups is 'Intel Core 2 Quad Q6600 vs. Nvidia GeForce GTX280/GTX285', where also only a single core of the Intel CPU is actually used. While the date of release of the Intel Core 2 Quad Q6600 CPU is 07.01.2007, the Nvidia GPUs were first available on 16.06.2008/15.01.2009. Obviously, there is a gap of at least one and a half year. Since Intel CPUs based on the current Intel Nehalem microarchitecture were also available at the end of 2008, we decided to consider these CPUs for benchmarks—in particular the Intel Core i7-920—. In fact, the Intel Core i7-920 performs about 1.5 times faster than the Intel Core 2 Quad Q6600, with its clock frequency being only higher by a factor of about 1.1.

For these reasons, we consider it justified to suggest almost all benchmark results, consciously ignoring these issues, to be overestimated by at least a factor 10−15.

7.2. What about the programming effort?

This question is rather subjective since the time that is necessary to create executable applications strongly depends on the skills of the programmer. Surprisingly, many people are able to write CUDA code in next to no time. On the other hand, it becomes difficult for them to do the same on the CPU.[6] Although this partly agrees with our experiences, we in contrast never felt, for instance, using OpenMP to be a book of seven seals. On the contrary, we deemed it one of the easiest ways to improve the performance of our CPU programs without doing more than just using some `#pragma` (preprocessor) directives within our codes, to put it simple.

However, on the part of CUDA there are very good tutorials by Nvidia itself [17, 18], and also plenty of CUDA codes to immediately become acquainted with the subject. To write first CUDA programs therefore takes only a few days, without being well grounded in parallel programming. We are sure that this is one of the reasons why so many people get on well with CUDA.

While writing simple CUDA programs is straightforward, optimizing them may require the programmer to be familiar with the details of the graphics hardware. Since the latter point may rapidly become very time-consuming, there should be always the question

[5] We assume that ATI/AMD GPUs have mechanisms to schedule threads, which are similar to those on Nvidia graphics boards.

[6] We discussed this point with other participants of a workshop on GPGPU: 'Simulations on GPU', 11 − 12 June 2009, Leipzig.

about the cost-benefit ratio. With respect to what was done over the course of the present work, it sometimes took us several hours to finalize optimizations of our CUDA codes, which then in turn performed about 2 times faster than the original (simple) implementations. Since optimizing (CUDA) programs is a learning process, relatively quickly we were able to recycle programming techniques for further CUDA projects.

As already noted, we experienced the same on the part of OpenMP programming for the CPU. While writing simple OpenMP programs is straightforward, tuning them to give best performance requires some experiences in parallel programming. Nonetheless, there are many tutorials on OpenMP programming. We therefore cannot understand people to not make use of this really simple framework, but on the other hand to spend a lot of time in optimizing CUDA code.

7.3. Reliability of GPGPU calculations

This section possibly concerns one of the most interesting point about GPGPU. While we had never any trouble with the accuracy of calculations—in Section 5, we had both the GPU and the CPU give exactly the same simulation results when using Ranlux—, we sometimes observed faulty estimates of system quantities when simulating large spin systems, which may be summarized as follows:

- Simulating the Ising model on two-dimensional lattices with extents $L \leq 128$, and three-dimensional lattices with extents $L \leq 48$ did not cause any problems. We used 16 Tesla C1060 graphics cards and 2 GeForce GTX285 graphics cards, with all of them producing exactly the same output for the same simulation setup. We also observed board temperatures on the GTX285, which reached maximum values of about 70°C.
- Having changed to $L = 256$ (in two dimensions) and $L = 64$ (in three dimensions), all simulations on GTX285 graphics cards gave absolutely wrong results,[7] and sometimes simulations on Tesla C1060 boards displayed the same behavior. In addition, all incorrect outputs were not reproducible. Since all graphics boards hitherto worked well, we tried to isolate the source of this behavior.
  After having checked the source code again and again, we adjusted the fan of the GTX's cooling system to permanently run at 100%, which in consequence reduced the board temperature from 75°C to 68°C. With this adjustment suddenly one of the two GTX285 boards gave correct results,[8] while the other GTX285 board still produced waste output.
  In a further step, we adjusted the clock frequencies of the GTX285 boards to values equal to those on Tesla C1060 boards,[9] that is, GPU core clock = 602MHz and

[7] 'Absolutely wrong' actually means absolutely wrong. All outputs made absolutely no sense.

[8] 'Correct' means exactly the same results as on the (correct working) Tesla boards, and also exactly the same results as on the CPU. For these tests we used the Ranlux random number generator.

[9] To do so, we used Coolbits. Under Linux, Coolbits can be activated via `nvidia-xconfig --cool-bits = 1` or by directly inserting the option "`Coolbits`" "`1`" into the `device` section of the `xorg.conf`. When there are multiple Nvidia GPUs, `nvidia-xconfig --enable-all-gpus` configures an X screen

memory clock = 800MHz (see Table 5.1). To put it simple, we manually underclocked the GTX285 boards. As a consequence, both boards performed simulations of the Ising model on lattices with extent $L = 256$ (and $L = 64$ respectively) with maximum board temperatures of about 62°C. Surprisingly, both GTX285s gave correct simulation results, even if simulations ran several hours.

Due to these observations, we conclude that there is an insufficient cooling on both the GTX285 (@ default clock) and the Tesla C1060. As the problems described above are more likely to appear if board temperatures increase, we will refer to them as 'thermal induced soft-errors'. Investigations on testing the GPU memory for hard- and soft-errors are detailed in [23].

Since Tesla C1060 boards equal their GTX285 counterparts in almost all points, except for the drastically reduced clock frequencies, Nvidia seems to know about this problem in a certain sense. For this reason, we do not really understand why, for instance, a Tesla S1070 computing system is made up of a total of 4 Tesla C1060 boards, installed within a single 1U rack-unit (see Figure 7.1). If our observations prove right, such installations encourage the overheating of the Tesla boards, which then in the worst case give incorrect output. It therefore is a debatable point whether current Tesla boards are suitable for HPC applications. The point is that the outcome of (scientific) calculations is expected to be correct and should not be subject to errors due to hardware faults or soft-errors.

Accordingly, GeForce GTX280/GTX285 graphics boards are only suitable for HPC applications to limit extent, unless their clock frequencies are lowered. Reducing the GTX280/GTX285's memory clock, or even using Tesla boards, results in a reduction of the memory bandwidth by a factor $1.5 - 1.6$, compared to the memory bandwidth of the GTX280/GTX285 at default clock (Table 5.1). The loss in memory bandwidth can make simulations execute longer by about a factor $1.5 - 1.6$, as observed for instance in Sections 4.3 and 5.1.3.

As a consequence, almost all speedups that result from measurements with GeForce GTX280/GTX285 boards are too large by a factor of about 1.5. Considering the arguments from Section 7.1, we assume almost all benchmark results that do not confirm with our performance issues to be overestimated by a factor 15−20.

We therefore want to re-emphasize that all benchmark results throughout this thesis are based on runtime measurements using Nvidia Tesla C1060 graphics cards only.

7.4. Nvidia's next generation GPUs—Fermi

While current CUDA-capable graphics devices are predominantly designed for graphics rendering, Nvidia's next generation GPUs, named 'Fermi', provide features that are indispensable for their application in the field of scientific computations. With respect

on every Nvidia GPU in the system and therefore allows to activate Coolbits on all of them. Since Coolbits is an X server module, it is only available when running X.

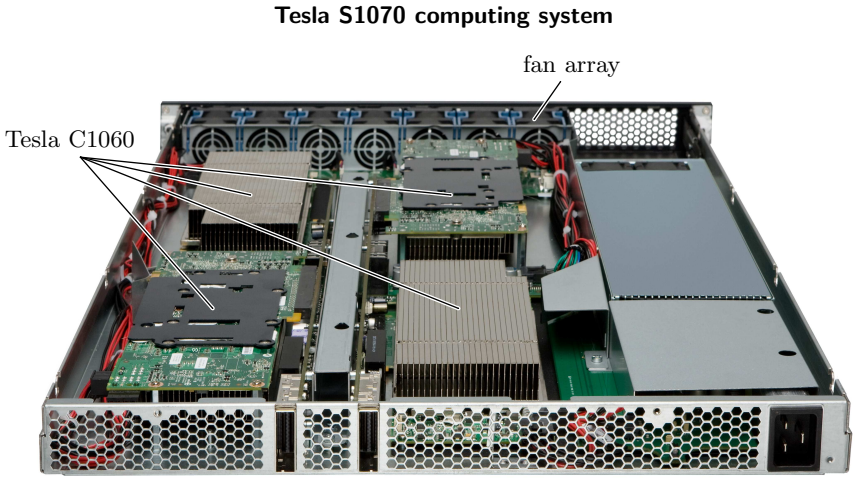

Figure 7.1.: Tesla S1070 computing system. 4 Nvidia Tesla C1060 graphics cards are installed within a single 1U rack unit. At the rear of the front panel there is an array of fans that are responsible for keeping all Tesla C1060 units at low temperatures.
Source: `http://www3.pny.com/Images/Tesla/Tesla_s1070_Rear_Elevated.jpg`.

to the previous section, the succeeding Tesla graphics cards (namely C2050/C2070) are equipped with ECC memory,[10] which allows to detect soft-errors and to correct them within certain limits. While Nvidia specifies Fermi's single-precision peak performance to be almost the same like that of the current generation, i.e. 1 TFLOP/s, its double-precision peak performance is more than 0.5 TFLOP/s. In fact, this is a gain of about a factor 7, compared to the Tesla C1060's double-precision peak performance. In addition, Nvidia incorporated the following improvements:[11]

- true cache hierarchy—allows to speedup algorithms that are unable to use the GPU's shared memory.
- larger shared memory—in particular, Nvidia enables the programmer to configure a total of 64kB on-chip memory (per multiprocessor) as 16kB first level cache and 48kB shared memory, or vice versa.
- dual warp scheduler—enables each multiprocessor to simultaneously schedule and dispatch instructions from two independent warps.
- concurrent kernel execution—the current Tesla generation matches only one kernel at the same time.
- full support for IEEE754-2008, 32-bit and 64-bit precision.
- 4 special function units per multiprocessor—currently, each multiprocessor has two special function unit.
- up to 20× faster atomic functions, compared to Tesla C1060.

[10] Fermi supports Single-Error Correct Double-Error Detect (SECDED) ECC codes that correct any single bit error in hardware as the data is accessed.
[11] `http://www.nvidia.com/object/fermi_architecture.html`

To sum up, Nvidia has designed its new Fermi architecture to give wide support for the needs of GPGPU users. Unfortunately, there is nothing known about the cooling solution. Of course, Fermi now supports ECC, but what about the decrease in performance if the rate of, for instance, thermal induced soft-errors becomes large?

8. Conclusion

In this thesis we investigated the applicability of GPGPU to spin models, such as the Ising model and the Ising spin glass. After having considered both of the two models from the theoretical point of view, and after having briefly introduced into the subject of simulating them by means of Monte Carlo methods, we were concerned with the CUDA programming model and the (Nvidia) graphics hardware itself. In a certain sense, GPUs (Graphics Processing Units) are akin to already existing SIMD machines as they process large amounts of data in a data parallel manner, similar to vector processors. On the other hand, they draw their immense computing power on scheduling an exceedingly large number of threads at the same time, each running its own subprogram concurrently to all the other threads in a thread parallel manner; Nvidia refers to this architecture as SIMT.

Since graphics cards are designed to predominantly perform SIMD-like operations, even though they are not restricted to do so, it seems most promising and maybe most natural to apply the CUDA programming model to local algorithms which allow to alter data objects independently of each other. With respect to spin models, we thus investigated the Ising model, as the simplest instance of a large class of Ising-like spin models, by means of the Metropolis algorithm. We observed speedups of about a factor $5 - 10$ for simulating the two-dimensional and the three-dimensional Ising model (in zero external magnetic field) on Tesla C1060 graphics cards instead of an Intel Core i7-920 quad-core x86 CPU—we used OpenMP to make our CPU implementations run on all 4 execution units of the CPU, and we optimized our CPU codes for SSE—. We also tried to port the Swendsen-Wang cluster algorithm for the Ising model to CUDA, unfortunately without any success. Although [24] presents a CUDA implementation of the Swendsen-Wang algorithm which is about 7.5 times faster than our parallelized CPU implementation of this algorithm, we deem GPGPU programming models being more suitable for parallelization of local algorithms.

In a further chapter, we investigated program execution times for simulating the Ising spin glass on Tesla C1060 graphics cards and an Intel Core i7-920, with outcomes having been almost the same as for the Ising model. All in all, we were able to attest the Tesla C1060 to perform simulations of Ising-like spin models about $5 - 10$ times faster than a current quad-core CPU.

Since doing Monte Carlo simulations on parallel computers suggest to also generate random numbers, as an essential part of any Monte Carlo simulation, in parallel, we ported the random number generators Ranlux and Mersenne Twister to CUDA. In addition we presented an alternative approach for efficiently producing parallel random numbers on Nvidia graphics cards. All generators were successfully tested for their quality. Depending on the random number generator, we observed speedups of up to one

order of magnitude for producing random numbers on the Tesla C1060 instead of the Core i7-920—we used parallel random number generators on the CPU, too—.

In a certain sense, our benchmark results seem to contradict somehow with what is presented in [21, 7, 24], who attest the GPU to absolutely outperform current quad-core CPUs. A closer look at what these authors—and many other authors of comparable papers—actually compare against each other brought out that almost all of their speedups follow from comparing highly optimized GPU codes, running on high-end graphics cards, with less optimized non-multithreaded CPU codes, running on legacy quad-core processors. In addition, many authors use consumer graphics cards for benchmarking, which are not suitable for scientific (HPC) applications as their clock frequencies are too high to have these boards reliably produce correct results for large simulations. For reasons that are discussed in Chapter 7, we assume such benchmark results to lead to speedups that are too large by a factor $15-20$. Throughout this thesis, we considered optimizations by means of OpenMP and SSE on the part of the CPU, and we used Tesla C1060 graphics cards which are designed for HPC applications. Our benchmark results therefore should give serious performance ratings for simulating Ising-like spin models on graphics cards.

In addition to assessing performance gains for simulating spin models on GPUs instead of CPUs, we were concerned with the reliability of GPGPU calculations. Since programmers expect the output of computer simulations to not suffer from hardware faults, it is of great importance for GPGPU to give reproducible and correct simulation results. During the course of simulating the Ising model on Tesla C1060 cards, we sometimes observed faulty program outputs which were not reproducible. Since we were not allowed to modify the Tesla C1060 hardware or even to change driver settings, we studied the reason for this behavior on two different Nvidia GeForce GTX285 graphics cards, which are almost equivalent to the Tesla C1060 boards, and also gave faulty program outputs for large simulations of the Ising model. We were able to link our observations to overheating of the graphics cards. By adjusting the fan speed of the GTX285's cooling system, we got one of the two GTX285 graphics cards give correct simulation results, while the other board still produced waste output. Lowering the clock frequencies, both GTX285 boards gave correct and reproducible output. Since the cooling system of Tesla C1060 graphics cards within a Tesla S1070 system is less performant than the GTX285's cooling system—Tesla C1060 cards within a Tesla S1070 system are cooled semi-passively (see Figure 7.1)—, we assume problems on those Tesla C1060 boards to also be the result of overheating. Since current Tesla cards do not support error correction mechanisms (ECC), it is a debatable point whether these cards are suitable for large simulations as simulation results are not guaranteed to be correct. The succeeding Nvidia Tesla graphics cards will support ECC on all memory layers which should change this point.

As the number of execution units on graphics cards continually grows, and, at least on the part of Nvidia, graphics cards will be increasingly designed for HPC applications, we expect them for a certain class of problems to become a powerful alternative to traditional cluster computers. Especially the fact that a single workstation that is equipped with some graphics cards of the succeeding Nvidia Tesla generation will provide several TFLOP/s of (double-precision) computing power is very amazing.

A. Exact solution of the 2D Ising model on finite squared lattices

As given in Section 1.2.1, the internal energy per spin computes as

$$\frac{E_{mn}}{mn} = -\frac{J}{mn}\frac{\mathrm{d}\ln Z_{mn}}{\mathrm{d}K} = -J\coth(2K) - \frac{J}{mn}\left[\sum_{i=1}^{4} Z_i'\right]\left[\sum_{i=1}^{4} Z_i\right]^{-1},$$

while the specific heat per spin is

$$\begin{aligned}\frac{(c_V)_{mn}}{k_{\mathrm{B}}mn} &= \frac{K^2}{mn}\frac{\mathrm{d}^2\ln Z_{mn}}{\mathrm{d}K^2} \\ &= -2K^2\,\mathrm{csch}^2(2K) + \frac{K^2}{mn}\left[\frac{\sum_{i=1}^{4} Z_i''}{\sum_{i=1}^{4} Z_i} - \left(\frac{\sum_{i=1}^{4} Z_i'}{\sum_{i=1}^{4} Z_i}\right)^2\right].\end{aligned}$$

To evaluate these equations, the following relations need to be used:

$$\begin{aligned}\gamma_0 &= 2K + \ln(\tanh(K))\,, & \gamma_0' &= 2(1+\mathrm{csch}(2K))\,,\\ \gamma_{r\neq 0} &= \ln(c_r + (c_r^2-1)^{1/2})\,, & \gamma_{r\neq 0}' &= c_r'(c_r^2-1)^{-1/2}\,,\\ c_r &= \cosh(2K)\coth(2K) - \cos(r\pi/n)\,, & c_r' &= 2\cosh(2K)(1-\mathrm{csch}^2(2K))\,,\end{aligned}$$

$$\begin{aligned}\gamma_0'' &= -4\,\mathrm{csch}(2K)\coth(2K)\,,\\ \gamma_{r\neq 0}'' &= c_r''(c_r^2-1)^{-1/2} - (c_r')^2 c_r(c_r^2-1)^{-3/2}\,,\\ c_r'' &= 8\,\mathrm{csch}^3(2K)\cosh^2(2K) + 4(\sinh(2K) - \mathrm{csch}(2K))\,,\end{aligned}$$

$$\begin{aligned}\frac{Z_1'}{Z_1} &= \frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r+1}'\tanh(\frac{m}{2}\gamma_{2r+1})\,,\\ \frac{Z_2'}{Z_2} &= \frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r+1}'\coth(\frac{m}{2}\gamma_{2r+1})\,,\\ \frac{Z_3'}{Z_3} &= \frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r}'\tanh(\frac{m}{2}\gamma_{2r})\,,\\ \frac{Z_4'}{Z_4} &= \frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r}'\coth(\frac{m}{2}\gamma_{2r})\,,\end{aligned}$$

$$
\begin{aligned}
\frac{Z_1''}{Z_1} &= \left[\frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r+1}'\tanh(\frac{m}{2}\gamma_{2r+1})\right]^2 \\
&\quad + \frac{m}{2}\sum_{r=0}^{n-1}[\gamma_{2r+1}''\tanh(\frac{m}{2}\gamma_{2r+1}) + \frac{m}{2}(\gamma_{2r+1}'\mathrm{sech}(\frac{m}{2}\gamma_{2r+1}))^2]\,, \\
\frac{Z_2''}{Z_2} &= \left[\frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r+1}'\coth(\frac{m}{2}\gamma_{2r+1})\right]^2 \\
&\quad + \frac{m}{2}\sum_{r=0}^{n-1}[\gamma_{2r+1}''\coth(\frac{m}{2}\gamma_{2r+1}) - \frac{m}{2}(\gamma_{2r+1}'\mathrm{csch}(\frac{m}{2}\gamma_{2r+1}))^2]\,, \\
\frac{Z_3''}{Z_3} &= \left[\frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r}'\tanh(\frac{m}{2}\gamma_{2r})\right]^2 \\
&\quad + \frac{m}{2}\sum_{r=0}^{n-1}[\gamma_{2r}''\tanh(\frac{m}{2}\gamma_{2r}) + \frac{m}{2}(\gamma_{2r}'\mathrm{sech}(\frac{m}{2}\gamma_{2r}))^2]\,, \\
\frac{Z_4''}{Z_4} &= \left[\frac{m}{2}\sum_{r=0}^{n-1}\gamma_{2r}'\coth(\frac{m}{2}\gamma_{2r})\right]^2 \\
&\quad + \frac{m}{2}\sum_{r=0}^{n-1}[\gamma_{2r}''\coth(\frac{m}{2}\gamma_{2r}) - \frac{m}{2}(\gamma_{2r}'\mathrm{csch}(\frac{m}{2}\gamma_{2r}))^2]\,.
\end{aligned}
$$

B. Real space renormalization and the finite size scaling method

This chapter introduces the subject of real space renormalization and the finite size scaling method. Both are of great relevance for investigating the critical behavior of many-particle systems in statistical physics.

B.1. Real space renormalization

As described in Section 1.4.1, the correlation length ξ diverges as $T \to T_c$, and as a consequence the number of degrees of freedom being correlated with each other becomes extremely large. In the framework of real space renormalization (see [5, 4]), summing over degrees of freedom that describe the short-range behavior of the system allows to reduce the number of degrees of freedom. Since near criticality only long-range interactions give significant contributions to system properties, the relevant physics will not change.

Summing over the short-range degrees of freedom can be done, for instance, by applying a *block spin transformation* $\mathcal{R}$, that is, to group b^d (d—spatial dimensionality) spins into blocks, and assign to each of these blocks a block spin s'_j. The way s'_j is achieved depends on the blocking method used. Subsequently, we will use the *majority rule*. The idea is to set s'_j to the spin value that is the most present one within the corresponding block. After this blocking procedure, all length scales need to be rescaled by a factor $1/b$ to obtain the same lattice spacing as before.

Applying a block spin transformation to a system at criticality will not change its correlation length,[1] which means that in the blocked system clusters of spins of all sizes occur, too. The critical system is said to be self-similar, or equivalently, it is invariant under block spin transformation (*scale invariance*).

To deduce the Hamiltonian $\mathcal{H}'$ of the blocked system from the Hamiltonian $\mathcal{H}$ of the unblocked system,[2] we introduce the projection operator T which implements the majority rule within each block (to avoid ambiguities, the number $n = b^d$ of spins per block is assumed to be odd):

$$T(s'_j; \{s_i : i \in \text{block}(j)\}) = \begin{cases} 1 & \text{if } s'_j \sum_i s_i > 0\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{B.1}$$

[1] At criticality the correlation length is infinite and therefore will not change when rescaling the system.

[2] In the following, $\mathcal{H}$ refers to the reduced Hamiltonian which absorbs the prefactor β within the exponential $\mathrm{e}^{-\beta H}$ into its parameters.

The Boltzmann factor of a certain spin configuration s' of the blocked system then results from summing over all Boltzmann factors of the unblocked system, each of them weighted by means of the projection operator (B.1):

$$\mathrm{e}^{-\mathcal{H}'(s')} = \sum_{\{s\}} \left(\mathrm{e}^{-\mathcal{H}(s)} \prod_{j=1}^{\text{blocks}} T(s'_j; \{s_i : i \in \text{block}(j)\}) \right). \tag{B.2}$$

Since $\sum_{s'_j} T(s'_j, \{s_i : i \in \text{block}(j)\}) = 1$, Eq. (B.2) leads to

$$\begin{aligned} Z' = \sum_{\{s'\}} \mathrm{e}^{-\mathcal{H}'(s')} &= \sum_{\{s'\}} \sum_{\{s\}} \left(\mathrm{e}^{-\mathcal{H}(s)} \prod_{j=1}^{\text{blocks}} T(s'_j; \{s_i : i \in \text{block}(j)\}) \right) \\ &= \sum_{\{s\}} \left(\mathrm{e}^{-\mathcal{H}(s)} \sum_{\{s'\}} \prod_{j=1}^{\text{blocks}} T(s'_j; \{s_i : i \in \text{block}(j)\}) \right) \\ &= \sum_{\{s\}} \left(\mathrm{e}^{-\mathcal{H}(s)} \prod_{j=1}^{\text{blocks}} \Big(\sum_{s'_j} T(s'_j; \{s_i : i \in \text{block}(j)\}) \Big) \right) = \sum_{\{s\}} \mathrm{e}^{-\mathcal{H}(s)} = Z \,, \end{aligned}$$

that is, the partition function is invariant under block spin transformation, and physics is left untouched.

Hereafter we will associate the Hamiltonian

$$\mathcal{H} = K_1 \sum_i s_i + K_2 \sum_{\langle ij \rangle} s_i s_j + K_3 \sum_{\langle\langle ij \rangle\rangle} s_i s_j + K_4 \sum_{i,j,k,l} s_i s_j s_k s_l + \ldots \tag{B.3}$$

with a vector $\{K\} = (K_1, K_2, \ldots)$ that is defined in some kind of parameter space, spanned by the couplings being present in the Hamiltonian.[3] In Eq. (B.3), $\sum_{\langle\langle ij \rangle\rangle} \ldots$ means the next-to-nearest-neighbor sum, $\sum_{i,j,k,l} \ldots$ refers to the plaquette contribution, and so forth. The block spin transformation $\mathcal{R}$ then maps the parameter vector $\{K\}$ onto another parameter vector $\{K'\} = \mathcal{R}(\{K\})$ within the same parameter space. The parameter space itself can be divided into the subspace of even couplings, which multiply interaction terms in the Hamiltonian that are invariant under spin flips $s \to -s$, and the subspace of odd couplings, with no mixing between them over the course of block spin transformations. The latter point makes the form of the Hamiltonian remain the same for all the time. Only the coupling parameters will change.

To be more general, hereafter the block spin transformation will be referred to as *renormalization group transformation* (RGT), but in fact nothing changes. The iteration of these RGTs leads to so-called *renormalization group flows* (RGF) within the parameter space. For a large set of models there are three *fixed points* $\{K^*\}$ the system evaluates to by repeated application of the RGT. These fixed points are defined by

$$\{K^*\} = \mathcal{R}(\{K^*\}) \,. \tag{B.4}$$

[3] Actually we assume the Hamiltonian $\mathcal{H}$ to include all possible spin-spin interactions. In the case of the Ising model the parameters are as follows: $K_1 = -\beta h$, $K_2 = -\beta J$ and $K_{i>2} = 0$.

Starting with a system with finite correlation length ξ, iterated RGTs lead to $\xi^{(n)} = b^{-n}\xi \overset{n\to\infty}{\longrightarrow} 0$, which is the case only for systems at very low temperature ($T \to 0$) or systems at very high temperature ($T \to \infty$). For these reasons, the corresponding fixed points are called *low-temperature fixed point* and *high-temperature fixed point*. Both are said to be attractive since all systems with finite correlation length will evaluate towards one of them. For $T = T_c$, the correlation length is left unchanged (i.e. $\xi = \infty$) under RGTs. As a consequence, the system remains at criticality, evaluating on the *critical surface*, which separates the low-temperature fixed point from the high-temperature fixed point, until it reaches the *critical fixed point*, referred to as $\{K^*\}$ from now onwards. $\{K^*\}$ is said to be a mixed fixed point since it is attractive within the critical surface and repulsive out of it.

To get information about the critical fixed point, we consider the RGT in the vicinity of $\{K^*\}$. For $\{K\} \approx \{K^*\}$ the RGT yields $\mathcal{R}(\{K\}) = \{K'\} \approx \{K^*\}$. Linearizing this transformation gives

$$K'_a - K^*_a \approx \sum_b T_{ab}\Big(K_b - K^*_b\Big)\,, \tag{B.5}$$

with $T_{ab} = \partial K'_a/\partial K_b|_{\{K\}=\{K^*\}}$ being the derivative of the RGT at the critical fixed point. With $\{\lambda^i\}$ being the eigenvalues of matrix $\boldsymbol{T}$, and $\{\phi^i\}$ being the corresponding left eigenvectors, i.e. $\sum_a \phi^i_a T_{ab} = \lambda^i \phi_b$, we define the scaling variables $u_i = \sum_a \phi^i_a (K_a - K^*_a)$. Applying an RGT to the scaling variables yields

$$\begin{aligned} u'_i = \sum_a \phi^i_a (K'_a - K^*_a) &= \sum_{a,b} \phi^i_a T_{ab} (K_b - K^*_b) \\ &= \sum_b \lambda^i \phi^i_b (K_b - K^*_b) = \lambda^i u_i = b^{y_i} u_i\,, \end{aligned} \tag{B.6}$$

where the $\{y_i\}$ are called *renormalization group eigenvalues*, which in turn are related to the critical exponents of the system. There are the following three possibilities:

- $y_i > 0$ — u_i is called a *relevant scaling variable*. Repeated RGTs drive u_i away from its fixed point value.
- $y_i < 0$ — u_i is called an *irrelevant scaling variable*. If starting close to the fixed point, u_i will iterate towards zero.
- $y_i = 0$ — u_i is called a *marginal scaling variable*. One cannot say, whether u_i will move away from the fixed point or towards it.

Subsequently, we will consider the class of the critical short-range Ising model. As suggested by Eq. (1.5), there will be two relevant scaling variables: a thermal scaling variable u_t with eigenvalue y_t, and a magnetic scaling variable u_h with eigenvalue y_h. Both of them may be referred to as some kind of 'knobs' adjustable by the experimentalist in order to bring the system near to criticality. In addition, there will be an infinite number of irrelevant scaling variables $u_3, \ldots$

Since the linearized RGT is valid only in the vicinity of the critical fixed point, the system has to be moved towards it. This can be done by applying a finite number of

RGTs, which at the same time render the irrelevant scaling variables vanish. On the part of the relevant scaling variables, we assume their values u_i to depend analytically on the deviations (t, h) of the original theory from its critical point. According to their definition, the scaling variables have to vanish for $\{K\} = \{K\}^*$, which makes them being of the form [5]

$$u_t = t/t_0 + \mathcal{O}(t^2, h^2)\,, \tag{B.7}$$

$$u_h = h/h_0 + \mathcal{O}(th)\,, \tag{B.8}$$

where t_0 and h_0 are non-universal constants.

To derive system quantities, the focus will be on the free energy density $f(\{K\}) = N^{-1} \ln Z$, with N being the number of lattice sites. As detailed, the partition function is invariant under RGTs, that is $Z = \sum_{\{s\}} \mathrm{e}^{-\mathcal{H}(s)} = \sum_{\{s'\}} \mathrm{e}^{-\mathcal{H}'(s')} = Z'$. Under renormalization the couplings $\{K\}$ change according to the RGF, but in addition, due to summing over a finite number of degrees of freedom, a constant term $Ng\{K\}$ appears in the free energy density of the blocked system [5]. Thus, $Z = Z'$ leads to $\exp\left[-Nf\{K\}\right] = \exp\left[-Ng(\{K\}) - N'f(\{K'\})\right]$, where $N' = b^{-d}N$. The free energy density therefore consists of an analytic part, given by $g(\{K\})$, which will not be of any interest when studying the critical behavior of the system, and a singular part, referred to as $f_s(\{K\})$. The singular part transforms as

$$f_s(\{K\}) = b^{-d} f_s(\{K'\}) \tag{B.9}$$

under RGT. Close to the fixed point, we write this in terms of scaling variables

$$f_s(u_t, u_h) = b^{-d} f_s(b^{y_t} u_t, b^{y_h} u_h) = b^{-nd} f_s(b^{ny_t} u_t, b^{ny_h} u_h)\,, \tag{B.10}$$

with all irrelevant scaling variables ignored. Since u_t and u_h grow under RGT, only a finite number n of transformations can be applied; otherwise the linearization of the RGT becomes invalid. We end the iteration at some point where $|b^{ny_t} u_t| = u_t^0$ with u_t^0 being arbitrary but fixed and small so that the linearization is still valid. Substituting Eqs. (B.7) and (B.8) into Eq. (B.10) yields

$$f_s(t, h) = |u_t/u_t^0|^{d/y_t} f_s\left(\pm u_t^0, \frac{u_h}{|u_t/u_t^0|^{y_h/y_t}}\right) = |t/t_0|^{d/y_t}\, \Phi\left(\frac{h/h_0}{|t/t_0|^{y_h/y_t}}\right). \tag{B.11}$$

With this equation, the scaling laws (1.17a)-(1.17f) can be deduced. The magnetic susceptibility χ, for instance, results from

$$\chi = \left.\frac{\partial^2 f_s}{\partial h^2}\right|_{h=0} = |t/t_0|^{d/y_t - 2y_h/y_t}\, \Phi_\chi(0) \propto |t|^{(d-2y_h)/y_t}\,.$$

Because of $\chi \propto |t|^{-\gamma}$, the critical exponent γ is

$$\gamma = \frac{2y_h - d}{y_t}\,. \tag{B.12}$$

The remaining critical exponents are given by

$$\alpha = 2 - \frac{d}{y_t}\,, \quad \beta = \frac{d - y_h}{y_t}\,, \quad \delta = \frac{y_h}{d - y_h}\,, \quad \nu = \frac{1}{y_t}\,, \quad \eta = d + 2 - 2y_h\,. \tag{B.13}$$

As can be seen, all critical exponents depend on the two renormalization group eigenvalues y_t and y_h, which in turn leads to relations between them [4]:

$$\begin{aligned} \text{Rushbrooke's law:} \quad & \alpha + 2\beta + \gamma = 2\,, & \text{Fisher's law:} \quad & (2 - \eta)\,\nu = \gamma\,, \\ \text{Griffiths' law:} \quad & \alpha + \beta(\delta + 1) = 2\,, & \text{Josephson's law:} \quad & \nu d = 2 - \alpha\,. \end{aligned} \tag{B.14}$$

B.2. The finite size scaling method

While scaling laws of the form $A \propto |t|^{-x}$ are valid only in the case of infinitely large systems, describing the critical behavior of finite systems leads to so-called finite size scaling. In the following, laws of the form

$$A(t, L) \propto L^{x/\nu}\, \Psi\!\left(\frac{t}{t_0} L^{1/\nu}\right) \tag{B.15}$$

will be referred to as *finite size scaling laws*, with Ψ being a *universal scaling function* that may differ for $t > 0$ and $t < 0$. In the limit $t \to 0$ the scaling function becomes a constant $\Psi(0)$, and thus is the same for all values of L.

Subsequently, it is assumed that near the critical fixed point the RGF is not affected by the finite size of the system. In the case of models with short-range interactions, such as the Ising model, this condition will be satisfied. Similar to infinite systems, repeated RGT will make the relevant scaling variables grow without any limit. For the same reasons as given above, we will need to end the iteration after n steps, where n can be defined by $L' = L/b^n$ reaching a constant value. This relation then allows to eliminate the factor b^n in Eq. (B.10). With Eqs. (B.7) and (B.8) we get

$$f_s(u_t, u_h, L) = b^{-nd} f_s(b^{n y_t} u_t, b^{n y_h} u_h, b^{-n} L) = L^{-d}\, \Psi\left(\frac{t}{t_0} L^{1/\nu}, \frac{h}{h_0} L^{y_h}\right) \tag{B.16}$$

for the singular part of the free energy density, but now as a function of the relevant scaling variables and the lattice extent L. From Eq. (B.16) we can deduce an expression for the magnetic susceptibility χ:

$$\chi = \left.\frac{\partial^2 f_s}{\partial h^2}\right|_{h=0} = L^{2y_h - d}\, \Psi_\chi\!\left(\frac{t}{t_0} L^{1/\nu}\right) = L^{\gamma/\nu}\, \Psi_\chi\!\left(\frac{t}{t_0} L^{1/\nu}\right). \tag{B.17}$$

B.3. Corrections to finite size scaling

Since up to now, all contributions from irrelevant scaling variables were systematically ignored, it will be necessary to remark (at least to know about it) that irrelevant scaling

variables will cause corrections to the scaling behavior figured out in this section.

In the following, we consider an irrelevant scaling variable, say, $u_3(t,h)$. While relevant scaling variables need to vanish on the critical surface,[4] irrelevant scaling variables may have $u_3(t=0,h=0)=u_3^0\neq 0$. Close to the critical fixed point, all terms other than u_3^0 can be ignored. With $u_3=u_3^0+\mathcal{O}(t,h^2)$, Eq. (B.16) then becomes [5]

$$f_s(u_t,u_h,L)=L^{-d}\,\Psi\left(\frac{t}{t_0}L^{1/\nu},\frac{h}{h_0}L^{y_h},u_3^0L^{-y_3}\right). \tag{B.18}$$

Assuming that Ψ is an analytic function of its arguments, the magnetic susceptibility, for instance, is

$$\chi=L^{\gamma/\nu}\,\Psi_\chi\left(\frac{t}{t_0}L^{1/\nu},u_3^0L^{-y_3}\right)\propto L^{\gamma/\nu}\left(1+A_1tL^{1/\nu}+A_2u_3^0L^{-y_3}+\dots\right). \tag{B.19}$$

Considering irrelevant scaling variables other than u_3 will give additional corrections to finite size scaling laws.

[4] Since relevant scaling variables grow under RGTs, the only way to let the system find its critical fixed point is to make all relevant scaling variables vanish on the critical surface.

C. CUDA—A closer look

The present chapter gives additional information on using CUDA. At first, we start with some remarks on the installation procedure and on how to compile CUDA programs. Next, we will go into some details concerning the CUDA programming model. Then we will consider performance issues in order to speedup CUDA programs.

C.1. Installing the CUDA environment

Before writing CUDA programs, the CUDA environment needs to be set up, which requires one to install the CUDA driver, the CUDA SDK (SDK—Software Development Kit), and the CUDA toolkit.[1] To execute CUDA programs on real graphics cards—there is also the possibility to run CUDA programs in some kind of emulation mode which uses the CPU—, a CUDA-capable graphics device is required. In principal CUDA works with all Nvidia GPUs from the G8x series onwards, whereas only the Quadro and the Tesla line should be chosen for high-performance computing.

Installing the CUDA driver on a (Debian) Linux system can be done as follows: Turn of the X Window System, e.g. execute `/sbin/init 1` as super user. Then run the driver package

```
$> sh cudadriver_*.run
```

and restart the X Window System, e.g. execute `/sbin/init 2`. To install the CUDA toolkit, run

```
$> sh cudatoolkit_*.run
```

as super user and accept the default installation path `/usr/local/cuda`. Afterwards adjust the environment variables `PATH` and `LD_LIBRARY_PATH`. For bash shell this can be done as follows:

```
$> export PATH=/usr/local/cuda/bin:$PATH
$> export LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH
```

When all is done, open a terminal and type in: `nvcc --version`. The output should hold some information about the `nvcc` compiler wrapper.

The CUDA SDK should be installed as a regular user. To do so, run

```
$> sh cudasdk_*.run
```

[1] The sources are available from **http://www.nvidia.de/object/cuda_get_de.html**.

and accept the default installation path `${HOME}/NVIDIA_GPU_COMPUTING_SDK`. When finished, enter the subdirectory `C/` and run `make`. The generated binaries will be installed under the home directory in `NVIDIA_GPU_Computing_SDK/C/bin/linux/release`. Enter this directory and type in

```
$> ./deviceQuery
```

The output should be of the following form (if there is a CUDA-capable device)

```
There is 1 device supporting CUDA

Device 0:  "Tesla C1060"
 CUDA Driver Version:  2.30
 CUDA Runtime Version:  2.30
 CUDA Capability Major revision number:  1
 CUDA Capability Minor revision number:  3
 ...

Test PASSED

Press ENTER to exit...
```

To compile CUDA programs, the `nvcc` compiler wrapper can be used. In the simplest case where the whole source code, including the `C` main function, is written into a single file, say, `testFile.cu`,

```
nvcc -o testProgram.x testFile.cu
```

generates an executable CUDA program.

C.2. The CUDA programming model

Since we already introduced into this subject, we restrict this section to only give further information to what was presented in Section 3.2.

As mentioned, the main point in writing CUDA programs is to detect computing tasks that can be outsourced for being processed on the graphics hardware. This directly leads to a set of functions being performed on the GPU, and a set of functions being executed on the CPU of the host system. To specify whether a function executes on the graphics card or the host's CPU, or whether it is callable by the graphics card itself or the host, CUDA introduces so-called *function type qualifiers* (see Table C.1) which have to be put in front of the declaration of any function.

These function type qualifiers enable the `nvcc` compiler wrapper to separate GPU code from that for the CPU, and so to pass them to appropriate compilers.

Other than for `__host__` functions, calling a `__global__` function requires to specify the number of CUDA threads that should execute the corresponding kernel. To do so, the `<<<grid,block>>>` syntax has to be used, with the first argument defining the

function type qualifier	executed on	callable by
`__host__`	host	host
`__global__`	graphics card	host
`__device__`	graphics card	graphics card

Table C.1.: CUDA function type qualifiers.

geometry of the grid of thread blocks, and the second argument defining the geometry of the thread blocks themselves. The grid is up to two-dimensional, and thread blocks are up to three-dimensional.

Both `grid` and `block` use the CUDA-specific `dim3` data type for their definition, as shown by the following lines:

```
// block configuration
dim3 block(block_x,block_y,block_z);
// grid configuration
dim3 grid(grid_x,grid_y);
```

On the part of the host, the values of the components of `grid` and `block` are accessable through `grid.X` (`X = x, y`) and `block.X` (`X = x, y, z`). Similarly, CUDA threads within a kernel request for these information using the `gridDim` and `blockDim` variables. To identify a certain CUDA thread within its block, or even a certain thread block within the grid of thread blocks, the `threadIdx` and `blockIdx` variables need to be used. With these information, each CUDA thread then can be related to a unique thread ID which in turn can be used to access certain data objects within a vector, a matrix, or a field.

To provide data objects on the graphics card, or even to get data from it, the host needs to be able to access some of the graphics card's memory layers. Since the graphics card is unable to act autonomously, it is also up to the host to allocate and free graphics memory. To dynamically allocate graphics memory

```
cudaMalloc((void **)&data,size);
```

can be used, where `cudaMalloc()` makes `data` point to the address of a field of `size` bytes, located in the graphics card's main memory. There are also other possibilities to dynamically allocate graphics memory [17], but in principal `cudaMalloc()` will be the common way. To free dynamically allocated graphics memory, `cudaFree(data)` has to be used. Using CUDA-specific *variable type qualifiers* (see Table C.2), graphics memory can be also allocated statically. For instance, the first two lines of

```
__device__ int intArray[128];
__constant__ int constIntArray[128];
__shared__ int *sharedPtr;
```

allocate two arrays of integer variables, each of size 128, the first (`intArray`) located in the graphics card's main memory, and the second (`constIntArray`) located in the constant memory. `sharedPtr` will be assigned to the shared memory at runtime.

variable type qualifier	writeable by	readable by
`__device__`	host, graphics card	host, graphics card
`__shared__`	graphics card	graphics card
`__constant__`	host	host, graphics card

Table C.2.: CUDA variable type qualifiers.

To exchange data between the host and the graphics card—again all data transfers need to be initiated by the host—, several copy functions, optimized for different data types, are available. All of these functions expect two pointers, usually addressing data residing on the one hand in the graphics card's memory, and on the other hand in the main memory of the host system. In addition, some information about the amount of the data that should be moved is necessary. For instance,

```
cudaMemcpy(dstPointer,srcPointer,size,copyDirection);
```

transfers a memory area of `size` bytes, pointed to by `srcPointer`, to a memory area pointed to by `dstPointer`, where `copyDirection` is one of

```
cudaMemcpyDeviceToHost,
cudaMemcpyHostToDevice,
cudaMemcpyHostToHost,
cudaMemcpyDeviceToDevice.
```

One has to consider that the main memory of the graphics card and the main memory of the host do not belong to the same physical address space. In a certain sense, this leads to a double entry bookkeeping, since all variables used by both the CPU and the GPU have to be allocated and managed twice.

To make some of these aspects more clear, the following listing utilizes some CUDA functions in order to calculate the sum of two (32×32)-matrices.

```
   #include <stdio.h>
   #include <limits.h>
   #include <cuda.h>

 5 #define X 32
   #define Y 32
   #define SIZE (X*Y)

   // declare matrices as one dimensional arrays.
10 static __device__ float A[SIZE];
   static __device__ float B[SIZE];
   static __device__ float C[SIZE];

   static __global__ void addMatrix()
15 {
    int
     tx=threadIdx.x,
     ty=threadIdx.y,
     bx=blockIdx.x,
20   by=blockIdx.y,
```

```
      // compute one dimensional matrix index
      id=(by*blockDim.y+ty)*gridDim.x*blockDim.x+bx*blockDim.x+tx;

     // add matrices A and B
25   // each CUDA thread computes exactly one element of matrix C
     C[id]=A[id]+B[id];
    }

    int main()
30  {
     int
      i,j;
     float
      matrix_A[SIZE],
35    matrix_B[SIZE],
      matrix_C[SIZE];
     // set matrix_A and matrix_B at (pseudo-) random
     srand(1);
     for(i=0;i<SIZE;i++){
40    matrix_A[i]=((float)rand())/INT_MAX;
      matrix_B[i]=((float)rand())/INT_MAX;
     }

     // copy matrix_A and matrix_B to the device's main memory (A,B)
45   cudaMemcpyToSymbol(A,matrix_A,SIZE*sizeof(float),0,
                        cudaMemcpyHostToDevice);
     cudaMemcpyToSymbol(B,matrix_B,SIZE*sizeof(float),0,
                        cudaMemcpyHostToDevice);

50   // specify block and grid configuration
     dim3 block(16,16);
     dim3 grid(X/block.x,Y/block.y);

     // make 32*32 CUDA threads run the kernel
55   addMatrix<<<grid,block>>>();

     // copy matrix C to matrix_C.
     cudaMemcpyFromSymbol(matrix_C,C,SIZE*sizeof(float),0,
                          cudaMemcpyDeviceToHost);
60
     // print matrix_C partially
     for(i=0;i<10;i++){
       printf("%f+%f=%f\n",matrix_A[i],matrix_B[i],matrix_C[i]);
     }
65
     return 0;
    }
```

Listing C.1: CUDA program that calculates the sum of two (32×32)-matrices, `matrix_A` and `matrix_B`. Both of the two matrices are set at (pseudo-) random, using the standard `C` random number generator `rand()`. Having copied the two matrices into the graphics card's main memory, the kernel `addMatrix()` calculates the sum of the two. Afterwards, the result is transfered to the host in order to print it partially.

Instead of exchanging data between the host and the graphics card by means of `cudaMemcpy()`, Listing C.1 uses `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`, which resolve the physical addresses of the statically allocated data objects `A`, `B` and `C`, located in the graphics card's main memory, and then copy the respective data areas from the host to the device, and vice versa.

Listing C.1 can be compiled and linked with the `nvcc` compiler wrapper which then produces an executable CUDA program.

C.3. Performance issues

This section is concerned with programming aspects/techniques which may significantly influence the performance of CUDA programs [17].

Instruction performance

As detailed in Section 3.3, each of the 30 multiprocessors of the Tesla C1060, or more precise their thread schedulers, manage the acting of thousands of CUDA threads in realtime with almost zero overhead. The necessity to instantiate a number of CUDA threads that is exceedingly larger than the number of physical execution units on the graphics card results from the circumstance that memory requests from the graphics card's non-cached main memory have to be hidden in such a way that the number of idle cycles of its execution units, due to waiting for requested data, is minimized. The multiprocessor's thread schedulers therefore switch between different warps in order to always have threads that are ready to be executed while other threads run idle. Processing warps means that the multiprocessor

- reads the instruction operands for each thread of the warp,
- executes the instruction,
- writes the result for each thread of the warp.

The effective instruction throughput strongly depends on the ratio of arithmetic operations per memory operation, and the number of active threads per multiprocessor.

To gain the instruction troughput, the programmer has to

- minimize the use of instructions that require lots of clock cycles,
- maximize the use of the available memory bandwidth for each category of memory,
- allow the thread scheduler to overlap memory transactions with mathematical computations as much as possible.

The first point can be attempted by using special CUDA-optimized functions such as `__[u]mul24()`, `__fmul()`, `__expf()`, `__cosf()`, ..., which perform up to one order of magnitude faster than their equivalents from `math.h`. Using these CUDA-optimized functions may lead to deviations from corresponding `math.h` functions. In the case of `__[u]mul24(x,y)`, for instance, only the 24 least significant bits of `x` and `y` are considered, which makes `__[u]mul24(x,y)` inapplicable for numbers larger than 2^{24}. Although similar restrictions also hold for CUDA-optimized functions other than `__[u]mul24()`, Nvidia attests almost all of them to be IEEE754 conform within certain limits.[2]

Even control flow instructions, such as `if` or `switch`, to name only a few of them, can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different execution paths are serialized, increasing the total number of instructions executed for

[2] As noted in Table 3.1, CUDA is not 100% IEEE754 conform which primarily results from not supporting all rounding modes stipulated by the IEEE754 specifications.

this warp. When all the different execution paths have completed, the threads converge back to the same execution path. Best performance can be achieved if the control flow only depends on the unique thread ID.

Memory performance

Requesting data from the graphics card's main memory results in a large number of clock cycles, necessary to process the corresponding memory access and to return the requested data objects. If there are enough CUDA threads, the thread schedulers are able to hide these memory requests by processing independent instructions while waiting for the memory accesses to complete. To make memory accesses affect the overall performance only a small fraction of time, a typical programming pattern is to stage data coming from the device's main memory into the shared memory, that is, to have each thread of a thread block to

- load data from device memory to shared memory,
- synchronize with all the other threads of the block, so that each thread can safely read shared memory locations that were written by different threads,
- process the data in shared memory,
- synchronize again if necessary to make sure that shared memory has been updated with the results,
- write the results back to device memory.

Another point concerns the access to the main memory itself. Since it is not cached, it is all the more important to follow the right memory access pattern to get maximum memory bandwidth. First, the graphics card is able to read 32-bit, 64-bit, or 128-bit words from its main memory in a single instruction. To have assignments such as

```
__device__ datatype deviceData[32];
datatype data = deviceData[threadID];
```

compile to a single load instruction, `datatype` must be such that `sizeof(datatype)` is equal to $4, 8$ or 16, and variables of type `datatype` must be aligned to `sizeof(datatype)` bytes, that is, have their address be a multiple of `sizeof(datatype)`. The alignment requirement is automatically fulfilled for built-in types, such as `int2`, `float2`, `float4`, ..., but can be also enforced by the compiler, using the alignment specifiers `__align__(4)`, `__align__(8)`, ...

Second, main memory bandwidth is used most efficiently when simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction. Coalescence is fulfilled as soon as the words accessed by all threads lie in the same segment of size equal to

32 bytes if all threads access 8-bit words,
64 bytes if all threads access 16-bit words,
128 bytes if all threads access 32-bit or 64-bit words.

Coalescing is also achieved for any pattern where multiple threads access the same address. Similar considerations also hold for shared memory access patterns.

D. Parallel random number generation—Implementations using CUDA

This chapter lists our implementations of the Ranlux random number generator and the CDAran32 random number generator, both using CUDA.

D.1. The Ranlux random number generator (single-precision)

Listing D.1 shows the entire source code of our implementation of the single-precision version of the Ranlux random number generator using CUDA.

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <math.h>
    #include <cuda.h>
 5  #include <cutil.h>
    #include <cutil_inline.h>

    #define MAX_INSTANCES 960
    #define RNGS_PER_BLOCK 4
10  #define MASK 0xffffff
    #define ONEBIT 0.59604644E-7

    static __device__ int RANLUX_d_instances;
    static __device__ int RANLUX_d_vec[4*MAX_INSTANCES][24];
15  static __device__ int RANLUX_d_carry[4*MAX_INSTANCES];
    static __device__ int RANLUX_d_pr[4*MAX_INSTANCES];
    static __device__ int RANLUX_d_prm[4*MAX_INSTANCES];
    static __device__ int RANLUX_d_ir[4*MAX_INSTANCES];
    static __device__ int RANLUX_d_jr[4*MAX_INSTANCES];
20  static __device__ int RANLUX_d_is[4*MAX_INSTANCES];
    static __device__ int RANLUX_d_isOld[4*MAX_INSTANCES];
    static __device__ __constant__ int RANLUX_d_next[24];

    #define STEP(pi1,pi2,pj1,pj2)\
25    d=(*pj1)-(*pi1)-(*carry);\
      (*pi1)=d&MASK;\
      (*pi2)+=(d<0);\
      d=(*pj2)-(*pi2);\
      (*carry)=(d<0);\
30    (*pi2)=d&MASK

    static __global__ void
    CDArlxs_subInit(const int *level,const int *seed)
    {
35   int
      i,k,l,
      ibit,jbit,xbit[31],
      ix,iy,
      *ptr,
40    bx=blockIdx.x,
```

```
      tx=threadIdx.x,
      myRNG=bx*blockDim.x+tx;

     if(myRNG<RANLUX_d_instances){
      if(level[myRNG]==0){
       for(i=0;i<4;i++){
        RANLUX_d_pr[4*myRNG+i]=109;
       }
      }else if(level[myRNG]==1){
       for(i=0;i<4;i++){
        RANLUX_d_pr[4*myRNG+i]=202;
       }
      }else if(level[myRNG]==2){
       for(i=0;i<4;i++){
        RANLUX_d_pr[4*myRNG+i]=397;
       }
      }

      i=seed[myRNG];

      for(k=0;k<31;k++){
       xbit[k]=i&1;
       i>>=1;
      }

      ibit=0;
      jbit=18;

      for(i=0;i<4;i++){
       ptr=&RANLUX_d_vec[4*myRNG+i][0];
       for(k=0;k<24;k++){
        ix=0;
        for(l=0;l<24;l++){
         iy=xbit[ibit];
         ix=(ix<<1)+iy;
         xbit[ibit]=(xbit[ibit]+xbit[jbit])&1;
         ibit=(ibit+1)%31;
         jbit=(jbit+1)%31;
        }
        if((k&3)==i){
         ix=16777215-ix;
        }
        ptr[k]=ix;
       }

       RANLUX_d_carry[4*myRNG+i]=0;
       RANLUX_d_ir[4*myRNG+i]=0;
       RANLUX_d_jr[4*myRNG+i]=7;
       RANLUX_d_is[4*myRNG+i]=23;
       RANLUX_d_isOld[4*myRNG+i]=0;
       RANLUX_d_prm[4*myRNG+i]=(RANLUX_d_pr[4*myRNG+i])%12;
      }
     }
    }

    void
    CDArlxs_init(const int *level,const int *seed,const int instances)
    {
     int
      i,
      *d_level,
      *d_seed,
      h_next[24];

     if(instances <= MAX_INSTANCES){
```

```
      cutilSafeCall(cudaMemcpyToSymbol(RANLUX_d_instances,&instances,\
                                sizeof(int),0,cudaMemcpyHostToDevice));
      cutilSafeCall(cudaMalloc((void **)&d_level,\
                                instances*sizeof(int)));
110   cutilSafeCall(cudaMalloc((void **)&d_seed,\
                                instances*sizeof(int)));
      cutilSafeCall(cudaMemcpy(d_level,level,instances*sizeof(int),\
                                cudaMemcpyHostToDevice));
      cutilSafeCall(cudaMemcpy(d_seed,seed,instances*sizeof(int),\
115                             cudaMemcpyHostToDevice));

      for(i=0;i<24;i++){
       h_next[i]=(i+1)%24;
      }
120   cutilSafeCall(cudaMemcpyToSymbol(RANLUX_d_next,h_next,24*\
                                sizeof(int),0,cudaMemcpyHostToDevice));

      dim3 block(64,1);
      dim3 grid (MAX_INSTANCES/block.x,1);
125
      CDArlxs_subInit<<<grid,block>>>(d_level,d_seed);

      cutilSafeCall(cudaFree(d_level));
      cutilSafeCall(cudaFree(d_seed));
130
     }else{
      printf("\n!!!error!!! only %d RNG instances instantiateable\n",\
             MAX_INSTANCES);
      exit(1);
135  }
    }

    static __device__ void
    CDArlxs_update(int *vec,int *carry,int *pr,int *prm,int *ir,int *jr,
140                int *is,int *isOld)
    {
     int
      k,kmax,d,
      *pmin,*pmax,
145   *pi1,*pi2,*pj1,*pj2;

     pmin=vec;
     pmax=vec+24;
     pi1=pmin+2*(*ir);
150  pi2=pi1+1;
     pj1=pmin+2*(*jr);
     pj2=pj1+1;
     kmax=(*pr);

155  for(k=0;k<kmax;k++){
      STEP(pi1,pi2,pj1,pj2);
      pi1+=2;
      pi2+=2;
      pj1+=2;
160   pj2+=2;
      if(pi1==pmax){
       pi1=pmin;
       pi2=pi1+1;
      }
165   if(pj1==pmax){
       pj1=pmin;
       pj2=pj1+1;
      }
     }
```

```
  (*ir)+=(*prm);
  (*jr)+=(*prm);
  if((*ir)>=12) (*ir)-=12;
  if((*jr)>=12) (*jr)-=12;
  (*is)=2*(*ir);
  (*isOld)=(*is);
}

static __device__ void
CDAranlxs_sub(float *r,const int n,const int myRNG)
{
 int
  i,
  tx=threadIdx.x,
  ty=threadIdx.y,
  carry=RANLUX_d_carry[4*myRNG+ty],
  pr=RANLUX_d_pr[4*myRNG+ty],
  prm=RANLUX_d_prm[4*myRNG+ty],
  ir=RANLUX_d_ir[4*myRNG+ty],
  jr=RANLUX_d_jr[4*myRNG+ty],
  is=RANLUX_d_is[4*myRNG+ty],
  isOld=RANLUX_d_isOld[4*myRNG+ty];

 __shared__ int vec[4*RNGS_PER_BLOCK][25];

 for(i=0;i<24;i++){
  vec[4*tx+ty][i]=RANLUX_d_vec[4*myRNG+ty][i];
 }

 for(i=0;i<n;i++){
  is=RANLUX_d_next[is];
  if(is == isOld){
   CDArlxs_update(vec[4*tx+ty],&carry,&pr,&prm,\
                  &ir,&jr,&is,&isOld);
  }
  r[4*i+ty]=ONEBIT*(float)(vec[4*tx+ty][is]);
 }

 for(i=0;i<24;i++) {
  RANLUX_d_vec[4*myRNG+ty][i]=vec[4*tx+ty][i];
 }

 RANLUX_d_carry[4*myRNG+ty]=carry;
 RANLUX_d_pr[4*myRNG+ty]=pr;
 RANLUX_d_prm[4*myRNG+ty]=prm;
 RANLUX_d_ir[4*myRNG+ty]=ir;
 RANLUX_d_jr[4*myRNG+ty]=jr;
 RANLUX_d_is[4*myRNG+ty]=is;
 RANLUX_d_isOld[4*myRNG+ty]=isOld;
}

static __global__ void
CDAranlxs(float *randomNumbers,const int numbersPerInstance)
{
 int
  tx=threadIdx.x,
  bx=blockIdx.x,
  myRNG=bx*blockDim.x+tx;
 float
  *ptr=&randomNumbers[myRNG*numbersPerInstance];

 CDAranlxs_sub(ptr,numbersPerInstance/4,myRNG);
}

void
```

```
    CDAranlxs_getNumbers(float *randomNumbers,
                         const int numbersPerInstance)
    {
     dim3 block(RNGS_PER_BLOCK,4);
240  dim3 grid(MAX_INSTANCES/block.x,1);

     CDAranlxs<<<grid,block>>>(randomNumbers,numbersPerInstance);
    }
```

Listing D.1: Entire source code implementing the single-precision version of the Ranlux random number generator using CUDA.

In lines 1-22, header files are included, symbolic constants are defined, and device memory is statically allocated. Note that the `static` prefixes in lines 13-22 declare the corresponding variables to be visible only within the corresponding file scope.

The macro `STEP()`, defined between lines 24 and 30, is used within the update kernel `CDArlxs_update()`. Successive execution of `STEP()` realizes the recursion (4.1).

In lines 32-136, the functions `CDArlxs_init()` and `CDArlxs_subInit()` are defined. Both implement the initializing process of the Ranlux instances, with the former one being callable from outside the corresponding file scope whereas the latter one is not—it is declared as `static`—. `CDArlxs_init()` acts as an interface function, passing its argument list to the kernel `CDArlxs_subInit()` which then is executed on the graphics card. In other words, `CDArlxs_init()` encapsulates the CUDA kernel `CDArlxs_subInit()`.

The succeeding lines until line 177 define the update kernel `CDArlxs_update()` which is of `__device__` type in order to be callable by the CUDA threads that execute the kernel `CDAranlxs_sub()`. The latter one is also of `__device__` type. It will be processed within the `__global__` kernel `CDAranlxs()`. Taking a closer look at `CDAranlxs_sub()`, we used a technique that is common practice in writing CUDA code:

- Line 194 defines the array `vec[0..15][0..24]` to reside in shared memory.
- In lines 196-198, each CUDA thread loads its random state, located in main memory, into the shared memory `vec[4*tx+ty][..]`.
- Lines 200-207 then iterate the update (`CDArlxs_update()`) of these random states as much as possible (this requires large values of `n`, the amount of requested random numbers) in order to make extensive use of `vec[..][..]`, and at the same time to avoid expensive memory calls during the generation of random numbers.
- In lines 209-219, data that changed during the update is restored to main memory.

To request for random numbers, `CDAranlxs_getNumbers()`, defined in lines 235-243, need to be used. It encapsulates the CUDA kernel `CDAranlxs()`, which then executes on the graphics card.

We also created an implementation of the double-precision version of Ranlux using CUDA. Implementations using the CPU were also created—we used OpenMP to make our codes perform on multi-core CPUs—.

D.2. The CDAran32 random number generator

Listing D.2 shows our implementation of the CDAran32 kernel which realizes Algorithm 3. For subsequent descriptions, we assume the number of CDAran32 instances to be 7680, where all these generators are organized into groups of 64 each. Generator configurations (b_1, b_2, b_3, a) as well as states (y, z) reside in the graphics card's main memory. They are accessable through the arrays d_RNG_configuration[0..89] of data type uint4, and d_RNG_states[0..7679] of data type uint2. The array d_RNG_mapping[0..7679], located in the graphics card's main memory, holds information about the mapping between CDAran32 instances and generator configurations.

To bijectively assign states (y, z) to the generator instances, the variable d_id_offset, located in the graphics card's constant memory, is used (see lines 10 and 18)—each time the CDAran32 kernel executes, d_id_offset is altered by using the function argument id_offset (line 34), which is assumed to be at random—.

The update kernel CDAran32_float(), which is consecutively executed in line 23/24, implements the recursion (4.5). CDAran32_float() is of __device__ type in order to be callable by the CUDA threads that execute CDAran32().

```
    static __global__ void
    CDAran32(float *random_numbers,const int count_per_RNG,
             const int id_offset)
    {
 5   int
      i,j,
      tx=threadIdx.x,
      bx=blockIdx.x,
      id=bx*blockDim.x+tx,
10    virtual_id=(id+d_id_offset)%7680,
      virtual_tx=tx;
     uint4
      configuration;

15   __shared__ uint2 state[64];

     configuration=d_RNG_configuration[d_RNG_mapping[id]];
     state[tx]=d_RNG_states[virtual_id];
     __syncthreads();
20
     for(i=0;i<count_per_RNG;i+=8){
      for(j=0;j<8;j++){
       random_numbers[id+(i+j)*7680]=\
       CDAran32_float(&configuration,&state[virtual_tx]);
25    }
      __syncthreads();
      virtual_tx=(virtual_tx+((state[0].x)&31))&63;
      __syncthreads();
     }
30
     d_RNG_states[virtual_id]=state[tx];

     if(id==0){
      d_id_offset=(d_id_offset+id_offset)%7680;
35   }
    }
```

Listing D.2: Implementation of the CDAran32 kernel.

As can be seen from Listing D.2, `CDAran32` expects a pointer (`random_numbers`) to an array that has to be filled with (7680 × `count_per_RNG`) single-precision random numbers. The meaning of the third argument `id_offset` was already addressed.

In lines 7-9, each CUDA thread calculates its (unique) thread ID. Line 15 defines the array `state[0..63]` to reside in the shared memory. Lines 17-19 then copy the thread's configuration as well as a randomly but bijectively chosen state (y, z) into local memory and shared memory, allowing for a fast access to them during the generation of random numbers. While each generator instance holds its exclusive generator configuration, i.e. it always loads the same configuration into local memory (line 17) each time the kernel is executed, the relation between generators and states is not fixed.

Line 19 makes all CUDA threads within the same thread block wait for all memory accesses to complete. Afterwards all CUDA threads start producing random numbers (lines 21-29). Over the course of generating `count_per_RNG` random samples per generator instance, CUDA threads within the same thread block interchange their random states each time after having written 8 random samples to `random_numbers` (line 23/24). To have this state-interchange procedure being at random, each CUDA thread uses a virtual thread ID `virtual_tx` for accessing random states `state[virtual_tx]`. `virtual_tx` initially equals its thread ID `threadIdx.x` (line 11), but is altered every 8 update steps using the random state `state[0].x` (line 27). To avoid race conditions, line 27 is surrounded by synchronization points (lines 26-28). Since all CUDA threads within the same thread block use the same value to alter their virtual thread ID, the mapping between CUDA threads and states is always bijective.

Having generated all requested random numbers, all states are restored to main memory, and the variable `d_id_offset` is modified (line 34) in order to request for states different from the present ones when executing the kernel the next time.

E. Simulating the Ising model— Implementations and simulation results

This chapter contains information about how to implement the checkerboard procedure for the Ising model using CUDA. We list system quantities E, c_V, $|m|$ and χ from simulating the two-dimensional and the three-dimensional Ising model by means of our CUDA and OpenMP implementations. We also list total per-replicum execution times of our simulations as well as the corresponding mean update times per spin.

E.1. Implementing the checkerboard procedure using CUDA

Listing E.1 shows the source code of our CUDA kernel that realizes the first update step of the checkerboard procedure for the two-dimensional Ising model.

```
    // lattice geometry
    #define LX 512
    #define LY 512
    #define NVOL (LX*LY)
 5  // thread block geometry
    #define SUB_X 64
    #define SUB_Y 2
    #define HALF_SUB_X (SUB_X/2)
    // zero external magnetic field?
10  #define ZERO_FIELD
    // double-precision?
    // #define DOUBLE
    #if defined DOUBLE
      #define FLOAT_TYPE double
15  #else
      #define FLOAT_TYPE float
    #endif

    static __global__ void
20  update_checkerboard_even(char *spin_field,
                             const FLOAT_TYPE *random_numbers)
    {
     int
      tx=threadIdx.x,
25    ty=threadIdx.y,
      bx=blockIdx.x,
      by=blockIdx.y,
      g_id,s_id,i_temp,sum_nn;
    #ifndef ZERO_FIELD
30   FLOAT_TYPE
      local_b_field,
      f_temp;
    #endif

35   __shared__ char sub_field[SUB_X*SUB_Y];
```

```
    g_id=by*SUB_X*SUB_Y*gridDim.x+ty*SUB_X*gridDim.x+bx*SUB_X+tx;
    s_id=ty*SUB_X+tx;

    sub_field[s_id]=spin_field[g_id];
    sub_field[s_id+HALF_SUB_X]=spin_field[g_id+HALF_SUB_X];

    __syncthreads();

    i_temp=(ty&1);
    g_id=g_id+tx+i_temp;
    s_id=s_id+tx+i_temp;

    // (+1)-direction
    if(i_temp==0)            sum_nn=sub_field[s_id+1];
    else{
     if(tx!=(HALF_SUB_X-1)) sum_nn=sub_field[s_id+1];
     else{
      if(bx==(gridDim.x-1)) sum_nn=spin_field[g_id-LX+1];
      else                  sum_nn=spin_field[g_id+1];
     }
    }
    // (-1)-direction
    if(i_temp==1)            sum_nn+=sub_field[s_id-1];
    else{
     if(tx!=0)               sum_nn+=sub_field[s_id-1];
     else{
      if(bx==0)              sum_nn+=spin_field[g_id+LX-1];
       else                  sum_nn+=spin_field[g_id-1];
     }
    }
    // (+2)-direction
    if(by==(gridDim.y-1)){
     if(ty==(blockDim.y-1)) sum_nn+=spin_field[g_id-NVOL+LX];
     else                   sum_nn+=sub_field[s_id+SUB_X];
    }else{
     if(ty==(blockDim.y-1)) sum_nn+=spin_field[g_id+LX];
     else                   sum_nn+=sub_field[s_id+SUB_X];
    }
    // (-2)-direction
    if(by==0){
     if(ty==0)              sum_nn+=spin_field[g_id+NVOL-LX];
     else                   sum_nn+=sub_field[s_id-SUB_X];
    }else{
     if(ty==0)              sum_nn+=spin_field[g_id-LX];
     else                   sum_nn+=sub_field[s_id-SUB_X];
    }

   #ifdef ZERO_FIELD
    i_temp=sum_nn*sub_field[s_id];
    if(i_temp<=0){
     spin_field[g_id]=__mul24(sub_field[s_id],-1);
    }else if(d_acceptance_ratios[i_temp]>random_numbers[g_id]){
     spin_field[g_id]=__mul24(sub_field[s_id],-1);
    }
   #else
    local_b_field=sum_nn+d_b_field;
    f_temp=d_minus_two_beta*local_b_field*sub_field[s_id];
   #ifdef DOUBLE
    f_temp=(double)exp(f_temp);
   #else
    f_temp=(float)expf(f_temp);
   #endif
    if(f_temp>=((FLOAT_TYPE)1.0)){
     spin_field[g_id]=__mul24(sub_field[s_id],-1);
    }else if(f_temp>random_numbers[g_id]){
```

```
      spin_field[g_id]=__mul24(sub_field[s_id],-1);
     }
    #endif
105 }
```

Listing E.1: CUDA kernel that realizes the first update step of the checkerboard procedure for the two-dimensional Ising model, as depicted in Figure 5.3.

Lines 1 and 7 define the lattice to be of geometry (512×512), and the sublattices to be of geometry (64×2). In order to avoid expensive memory calls during the update procedure, we copy entire sublattices into the shared memory, which happens in lines 40 and 41. Since the total number of CUDA threads that execute this kernel is only half the number of the lattice sites, copying sublattices into shared memory requires each thread to request for two spin values. To ensure that all threads finished their copy processes, line 43 defines a barrier (`__syncthreads()`). Lines 43-45 then make each CUDA thread point to spins that are located on even lattice sites, located in both the shared memory (`s_id`) and the main memory (`g_id`). In lines 49-104, each CUDA thread calculates the sum over its nearest-neighbor spins, and if necessary it flips its own spin. If spins change their orientation, their new values are written to the main memory.

Our implementation considers both zero external magnetic field and non-zero magnetic field. It therefore introduces the symbolic constant `ZERO_FIELD` which, if defined, makes the code use precomputed acceptance ratios `d_acceptance_ratios[]`, stored in the constant memory of the graphics card. If `ZERO_FIELD` is not defined, the exponential $\exp\{-\beta(E_\nu - E_\mu)\}$ is evaluated. The implementation also distinguish between calculations with single-precision (line 97) and double-precision (line 93). If the symbolic constant `DOUBLE` is not defined, calculations concerning the update of spins are performed with single-precision.

In addition to the present update kernel there is also a kernel that updates all odd lattice sites in exactly the same way. Even there are kernels that take measurements.

We also created implementations of the checkerboard procedure for the three-dimensional Ising model as well as implementations for the CPU using OpenMP to make the code perform on multi-core CPUs.

E.2. Simulation results—Metropolis algorithm

The following tables list system quantities E, c_V, $|m|$ and χ from simulating the two-dimensional and the three-dimensional Ising model in zero external magnetic field by means of the checkerboard procedure. We considered lattices with periodic boundary conditions and different extents L as well as different random number generators. For each combination $(L, \text{random number generator})$ a total of 10 replica, each containing $1000000 = 10^6$ Monte Carlo estimates of E, c_V, $|m|$ and χ, were generated. All these quantities and their statistical errors were evaluated by means of the `UWerr` script [26]. Since the errors of the errors were much smaller than 1%, we assume all statistical errors to be accurate. In addition to Monte Carlo estimates and their errors, we also list integrated autocorrelation times.

2D Ising model—internal energy E per spin, specific heat c_V per spin					
L	Generator	E	c_V	$\tau_{\mathrm{int},E}$	τ_{int,c_V}
16	exact	−1.45306485...	1.49870495...		
	pranlxs|0	−1.45315(7)	1.4991(7)	0.7980(16)	0.6221(10)
	CDAranlxs|0	−1.45315(7)	1.4991(7)	0.7980(16)	0.6221(10)
	pmts	−1.45303(7)	1.4996(7)	0.7969(16)	0.6238(10)
	CDAmts	−1.45312(7)	1.4984(7)	0.7937(16)	0.6206(10)
	CDAran32	−1.45306(7)	1.4984(7)	0.7966(16)	0.6224(10)
	pranlxd|1	−1.45314(7)	1.4993(7)	0.7986(16)	0.6217(10)
	CDAranlxd|1	−1.45314(7)	1.4993(7)	0.7986(16)	0.6217(10)
	pmtd	−1.45310(7)	1.4998(7)	0.7985(16)	0.6218(10)
	CDAmtd	−1.45308(7)	1.4986(7)	0.7956(16)	0.6225(10)
	CDAran64	−1.45302(7)	1.4989(7)	0.7979(16)	0.6239(10)
32	exact	−1.43365846...	1.84676759...		
	pranlxs|0	−1.43364(4)	1.8463(9)	1.149(3)	0.6995(16)
	CDAranlxs|0	−1.43364(4)	1.8463(9)	1.149(3)	0.6995(16)
	pmts	−1.43359(5)	1.8476(9)	1.149(3)	0.6961(16)
	CDAmts	−1.43367(5)	1.8463(9)	1.153(3)	0.6994(16)
	CDAran32	−1.43362(5)	1.8481(9)	1.149(3)	0.6972(16)
	pranlxd|1	−1.43365(5)	1.8471(9)	1.155(3)	0.7020(16)
	CDAranlxd|1	−1.43365(5)	1.8471(9)	1.155(3)	0.7020(16)
	pmtd	−1.43366(5)	1.8457(9)	1.148(3)	0.6989(16)
	CDAmtd	−1.43374(5)	1.8452(9)	1.142(3)	0.6981(16)
	CDAran64	−1.43365(5)	1.8475(9)	1.146(3)	0.6993(16)
64	exact	−1.42393838...	2.19221139...		
	pranlxs|0	−1.42392(3)	2.1922(12)	1.881(7)	0.850(3)
	CDAranlxs|0	−1.42392(3)	2.1922(12)	1.881(7)	0.850(3)
	pmts	−1.42394(3)	2.1921(12)	1.885(7)	0.851(3)
	CDAmts	−1.42389(3)	2.1924(12)	1.896(7)	0.851(3)
	CDAran32	−1.42394(3)	2.1916(12)	1.881(7)	0.848(3)
	pranlxd|1	−1.42390(3)	2.1920(12)	1.885(7)	0.851(3)
	CDAranlxd|1	−1.42390(3)	2.1920(12)	1.885(7)	0.851(3)
	pmtd	−1.42399(3)	2.1913(12)	1.871(7)	0.847(3)
	CDAmtd	−1.42393(3)	2.1947(12)	1.888(7)	0.851(3)
	CDAran64	−1.42396(3)	2.1919(12)	1.879(7)	0.852(3)
128	exact	−1.41907627...	2.53633133...		
	pranlxs|0	−1.41910(2)	2.5363(15)	2.741(18)	0.985(5)

Continued on the next page ...

	CDAranlxs|0	−1.41910(2)	2.5363(15)	2.741(18)	0.985(5)
	pmts	−1.41904(2)	2.5373(15)	2.759(18)	0.979(5)
	CDAmts	−1.41908(2)	2.5358(16)	2.742(18)	0.990(5)
	CDAran32	−1.41906(2)	2.5381(16)	2.775(18)	0.994(5)
	pranlxd|1	−1.41910(2)	2.5349(15)	2.764(18)	0.987(5)
	CDAranlxd|1	−1.41910(2)	2.5349(15)	2.764(18)	0.987(5)
	pmtd	−1.41908(2)	2.5362(15)	2.747(18)	0.982(5)
	CDAmtd	−1.41905(2)	2.5364(15)	2.735(18)	0.986(5)
	CDAran64	−1.41911(2)	2.5369(15)	2.774(18)	0.986(5)
256	exact	−1.41664495...	2.87978625...		
	pranlxs|0	−1.416652(15)	2.8775(21)	5.22(6)	1.405(13)
	CDAranlxs|0	−1.416652(15)	2.8775(21)	5.22(6)	1.405(13)
	pmts	−1.416610(15)	2.8832(21)	5.21(6)	1.385(12)
	CDAmts	−1.416654(15)	2.8777(21)	5.24(6)	1.401(12)
	CDAran32	−1.416627(15)	2.8823(21)	5.22(6)	1.425(13)
	pranlxd|1	−1.416619(15)	2.8837(21)	5.16(6)	1.389(12)
	CDAranlxd|1	−1.416619(15)	2.8837(21)	5.16(6)	1.389(12)
	pmtd	−1.416637(15)	2.8822(21)	5.22(6)	1.408(13)
	CDAmtd	−1.416618(15)	2.8852(21)	5.14(6)	1.412(13)
	CDAran64	−1.416650(15)	2.8797(21)	5.12(6)	1.409(13)

Table E.1.: Monte Carlo estimates of the internal energy per spin and the specific heat per spin from simulating the two-dimensional Ising model by means of the Metropolis algorithm on squared lattices with extent L. We also considered different random number generators.

2D Ising model—abs. magnet. $\|m\|$ per spin, magnetic susceptibility χ per spin						
L	Generator	$\|m\|$	χ	$\chi/(\beta L^2)$	$\tau_{\mathrm{int},\|m\|}$	$\tau_{\mathrm{int},\chi}$
16	pranlxs|0	0.71345(9)	61.535(11)	0.54545(10)	1.130(3)	1.023(2)
	CDAranlxs|0	0.71345(9)	61.535(11)	0.54545(10)	1.130(3)	1.023(2)
	pmts	0.71338(9)	61.521(11)	0.54533(10)	1.126(3)	1.021(2)
	CDAmts	0.71360(9)	61.551(11)	0.54559(10)	1.126(3)	1.020(2)
	CDAran32	0.71347(9)	61.534(11)	0.54543(10)	1.125(3)	1.022(2)
	pranlxd|1	0.71350(9)	61.541(11)	0.54550(10)	1.130(3)	1.026(2)
	CDAranlxd|1	0.71350(9)	61.541(11)	0.54550(10)	1.130(3)	1.026(2)
	pmtd	0.71347(9)	61.536(11)	0.54545(10)	1.128(3)	1.024(2)
	CDAmtd	0.71345(9)	61.535(11)	0.54545(10)	1.127(3)	1.022(2)
	CDAran64	0.71331(9)	61.518(11)	0.54530(10)	1.131(3)	1.025(2)
32	pranlxs|0	0.65435(12)	207.11(5)	0.45895(12)	2.186(8)	1.914(7)
	CDAranlxs|0	0.65435(12)	207.11(5)	0.45895(12)	2.186(8)	1.914(7)

Continued on the next page ...

	pmts	0.65422(12)	207.05(5)	0.45882(12)	2.176(8)	1.908(7)
	CDAmts	0.65436(12)	207.12(5)	0.45899(12)	2.199(8)	1.924(7)
	CDAran32	0.65423(12)	207.05(5)	0.45883(12)	2.177(8)	1.910(7)
	pranlxd|1	0.65431(12)	207.10(5)	0.45893(12)	2.195(8)	1.924(7)
	CDAranlxd|1	0.65431(12)	207.10(5)	0.45893(12)	2.195(8)	1.924(7)
	pmtd	0.65435(12)	207.12(5)	0.45897(12)	2.184(8)	1.911(7)
	CDAmtd	0.65454(12)	207.20(5)	0.45915(12)	2.177(8)	1.904(7)
	CDAran64	0.65437(12)	207.12(5)	0.45897(12)	2.180(8)	1.911(7)
64	pranlxs|0	0.59999(16)	696.6(3)	0.38593(14)	4.69(3)	4.04(2)
	CDAranlxs|0	0.59999(16)	696.6(3)	0.38593(14)	4.69(3)	4.04(2)
	pmts	0.60007(16)	696.8(3)	0.38602(14)	4.68(3)	4.03(2)
	CDAmts	0.59978(16)	696.3(3)	0.38576(14)	4.70(3)	4.06(2)
	CDAran32	0.60009(16)	696.8(3)	0.38603(14)	4.71(3)	4.06(2)
	pranlxd|1	0.59992(16)	696.5(3)	0.38587(14)	4.68(3)	4.06(2)
	CDAranlxd|1	0.59992(16)	696.5(3)	0.38587(14)	4.68(3)	4.06(2)
	pmtd	0.60036(16)	697.3(3)	0.38628(14)	4.69(3)	4.05(2)
	CDAmtd	0.60005(16)	696.8(3)	0.38601(14)	4.71(3)	4.06(2)
	CDAran64	0.60017(16)	697.0(3)	0.38615(14)	4.71(3)	4.07(2)
128	pranlxs|0	0.55050(19)	2345.1(1.1)	0.32480(16)	8.28(9)	7.09(7)
	CDAranlxs|0	0.55050(19)	2345.1(1.1)	0.32480(16)	8.28(9)	7.09(7)
	pmts	0.54997(19)	2341.7(1.2)	0.32433(16)	8.34(9)	7.20(7)
	CDAmts	0.55031(19)	2343.9(1.1)	0.32463(16)	8.27(9)	7.08(7)
	CDAran32	0.55020(19)	2343.4(1.2)	0.32456(16)	8.42(9)	7.22(7)
	pranlxd|1	0.55052(19)	2345.4(1.1)	0.32484(16)	8.28(9)	7.14(7)
	CDAranlxd|1	0.55052(19)	2345.4(1.1)	0.32484(16)	8.28(9)	7.14(7)
	pmtd	0.55029(19)	2343.8(1.1)	0.32462(16)	8.16(9)	7.01(7)
	CDAmtd	0.55013(19)	2342.7(1.1)	0.32446(16)	8.19(9)	7.08(7)
	CDAran64	0.55047(19)	2345.1(1.2)	0.32480(16)	8.55(9)	7.27(7)
256	pranlxs|0	0.50478(26)	7888(6)	0.27313(20)	18.6(4)	15.9(3)
	CDAranlxs|0	0.50478(26)	7888(6)	0.27313(20)	18.6(4)	15.9(3)
	pmts	0.50400(26)	7871(6)	0.27252(20)	18.6(4)	16.0(3)
	CDAmts	0.50493(26)	7890(6)	0.27318(20)	18.7(4)	16.0(3)
	CDAran32	0.50418(27)	7876(6)	0.27270(20)	19.1(4)	16.1(3)
	pranlxd|1	0.50411(26)	7872(6)	0.27257(20)	18.3(4)	15.8(3)
	CDAranlxd|1	0.50411(26)	7872(6)	0.27257(20)	18.3(4)	15.8(3)
	pmtd	0.50450(26)	7882(6)	0.27290(20)	18.8(4)	16.0(3)
	CDAmtd	0.50417(26)	7874(6)	0.27263(20)	18.7(4)	15.8(3)
	CDAran64	0.50472(26)	7887(6)	0.27309(20)	18.1(4)	15.6(3)

Table E.2.: Monte Carlo estimates of the absolute magnetization per spin and the magnetic susceptibility per spin from simulating the two-dimensional Ising model by means of the Metropolis algorithm on squared lattices with extent L. We also considered different random number generators.

3D Ising model—internal energy E per spin, specific heat c_V per spin					
L	Generator	E	c_V	$\tau_{\mathrm{int},E}$	τ_{int,c_V}
16	pranlxs|0	−1.03451(4)	1.7998(8)	1.060(2)	0.5391(8)
	CDAranlxs|0	−1.03451(4)	1.7998(8)	1.060(2)	0.5391(8)
	pmts	−1.03448(4)	1.8008(8)	1.060(2)	0.5403(8)
	CDAmts	−1.03450(4)	1.8009(8)	1.062(2)	0.5408(8)
	CDAran32	−1.03449(4)	1.8014(8)	1.064(2)	0.5412(8)
	pranlxd|1	−1.03448(4)	1.7995(8)	1.058(2)	0.5398(8)
	CDAranlxd|1	−1.03448(4)	1.7995(8)	1.058(2)	0.5398(8)
	pmtd	−1.03458(4)	1.8009(8)	1.059(2)	0.5408(8)
	CDAmtd	−1.03444(4)	1.8000(8)	1.061(2)	0.5402(8)
	CDAran64	−1.03443(4)	1.7994(8)	1.061(2)	0.5401(8)
24	pranlxs|0	−1.01535(3)	2.0514(9)	1.131(3)	0.5486(8)
	CDAranlxs|0	−1.01535(3)	2.0514(9)	1.131(3)	0.5486(8)
	pmts	−1.01533(3)	2.0507(9)	1.132(3)	0.5490(8)
	CDAmts	−1.01531(3)	2.0494(9)	1.130(3)	0.5484(8)
	CDAran32	−1.01530(8)	2.0480(9)	1.122(3)	0.5480(8)
	pranlxd|1	−1.01530(3)	2.0500(9)	1.127(3)	0.5492(8)
	CDAranlxd|1	−1.01530(3)	2.0500(9)	1.127(3)	0.5492(8)
	pmtd	−1.01530(3)	2.0495(9)	1.128(3)	0.5490(8)
	CDAmtd	−1.01531(3)	2.0497(9)	1.130(3)	0.5490(8)
	CDAran64	−1.01528(3)	2.0487(9)	1.126(3)	0.5496(8)
32	pranlxs|0	−1.006992(18)	2.2337(10)	1.170(3)	0.5544(9)
	CDAranlxs|0	−1.006992(18)	2.2337(10)	1.170(3)	0.5544(9)
	pmts	−1.006997(18)	2.2343(10)	1.170(3)	0.5535(9)
	CDAmts	−1.006998(18)	2.2347(10)	1.168(3)	0.5553(9)
	CDAran32	−1.006997(18)	2.2334(10)	1.172(3)	0.5553(9)
	pranlxd|1	−1.006996(18)	2.2333(10)	1.169(3)	0.5537(9)
	CDAranlxd|1	−1.006996(18)	2.2333(10)	1.169(3)	0.5537(9)
	pmtd	−1.007003(18)	2.2354(10)	1.166(3)	0.5543(9)
	CDAmtd	−1.006969(18)	2.2344(10)	1.168(3)	0.5534(9)
	CDAran64	−1.006992(18)	2.2344(10)	1.171(3)	0.5542(9)
48	pranlxs|0	−0.999742(12)	2.5077(12)	1.593(5)	0.6137(11)
	CDAranlxs|0	−0.999742(12)	2.5077(12)	1.593(5)	0.6137(11)
	pmts	−0.999719(12)	2.5039(12)	1.604(5)	0.6155(11)
	CDAmts	−0.999725(12)	2.5036(12)	1.594(5)	0.6152(11)
	CDAran32	−0.999719(12)	2.5053(12)	1.602(5)	0.6165(11)
	pranlxd|1	−0.999750(12)	2.5062(12)	1.595(5)	0.6143(11)

Continued on the next page ...

	CDAranlxd|1	−0.999750(12)	2.5062(12)	1.595(5)	0.6143(11)
	pmtd	−0.999755(12)	2.5067(12)	1.600(5)	0.6155(11)
	CDAmtd	−0.999742(12)	2.5051(12)	1.594(5)	0.6148(11)
	CDAran64	−0.999731(12)	2.5057(12)	1.590(5)	0.6151(11)
64	pranlxs|0	−0.996583(9)	2.7050(14)	2.024(7)	0.6831(14)
	CDAranlxs|0	−0.996583(9)	2.7050(14)	2.024(7)	0.6831(14)
	pmts	−0.996591(9)	2.7079(14)	2.012(7)	0.6808(13)
	CDAmts	−0.996588(9)	2.7033(14)	2.021(7)	0.6817(14)
	CDAran32	−0.996577(9)	2.7044(14)	2.020(7)	0.6827(14)
	pranlxd|1	−0.996570(9)	2.7046(14)	2.013(7)	0.6849(14)
	CDAranlxd|1	−0.996570(9)	2.7046(14)	2.013(7)	0.6849(14)
	pmtd	−0.996604(9)	2.7077(14)	2.018(7)	0.6810(14)
	CDAmtd	−0.996607(9)	2.7062(14)	2.017(7)	0.6801(14)
	CDAran64	−0.996590(9)	2.7060(14)	2.029(7)	0.6813(14)

Table E.3.: Monte Carlo estimates of the internal energy per spin and the specific heat per spin from simulating the three-dimensional Ising model by means of the Metropolis algorithm on cubic lattice with extent L. We also considered different random number generators.

3D Ising model—abs. magnet. $\|m\|$ per spin, magnetic susceptibility χ per spin						
L	Generator	$\|m\|$	χ	$\chi/(\beta L^2)$	$\tau_{\mathrm{int},\|m\|}$	$\tau_{\mathrm{int},\chi}$
16	pranlxs|0	0.26355(7)	77.62(3)	1.3679(5)	1.408(4)	1.308(3)
	CDAranlxs|0	0.26355(7)	77.62(3)	1.3679(5)	1.408(4)	1.308(3)
	pmts	0.26347(7)	77.59(3)	1.3674(5)	1.405(4)	1.306(3)
	CDAmts	0.26347(7)	77.60(3)	1.3675(5)	1.407(4)	1.309(3)
	CDAran32	0.26351(7)	77.61(3)	1.3677(5)	1.409(4)	1.310(3)
	pranlxd|1	0.26350(7)	77.60(3)	1.3676(5)	1.408(4)	1.308(3)
	CDAranlxd|1	0.26350(7)	77.60(3)	1.3676(5)	1.408(4)	1.308(3)
	pmtd	0.26363(7)	77.66(3)	1.3687(5)	1.404(4)	1.305(3)
	CDAmtd	0.26342(7)	77.57(3)	1.3670(5)	1.408(4)	1.308(3)
	CDAran64	0.26344(7)	77.57(3)	1.3670(5)	1.406(4)	1.307(3)
24	pranlxs|0	0.21408(6)	173.23(7)	1.3569(6)	1.580(5)	1.463(4)
	CDAranlxs|0	0.21408(6)	173.23(7)	1.3569(6)	1.580(5)	1.463(4)
	pmts	0.21402(6)	173.15(7)	1.3562(6)	1.583(5)	1.465(4)
	CDAmts	0.21401(6)	173.13(7)	1.3561(6)	1.580(5)	1.464(4)
	CDAran32	0.21403(6)	173.12(7)	1.3560(6)	1.571(5)	1.455(4)
	pranlxd|1	0.21401(6)	173.12(7)	1.3560(6)	1.573(5)	1.457(4)
	CDAranlxd|1	0.21401(6)	173.12(7)	1.3560(6)	1.573(5)	1.457(4)
	pmtd	0.21399(6)	173.09(7)	1.3558(6)	1.479(5)	1.462(4)

Continued on the next page ...

	CDAmtd	0.21400(6)	173.12(7)	1.3560(6)	1.582(5)	1.465(4)
	CDAran64	0.21392(6)	173.02(7)	1.3552(6)	1.569(5)	1.454(4)
32	pranlxs|0	0.18445(5)	305.26(13)	1.3449(6)	1.692(5)	1.566(5)
	CDAranlxs|0	0.18445(5)	305.26(13)	1.3449(6)	1.692(5)	1.566(5)
	pmts	0.18450(5)	305.26(13)	1.3454(6)	1.690(5)	1.565(5)
	CDAmts	0.18441(5)	305.16(13)	1.3449(6)	1.685(5)	1.559(5)
	CDAran32	0.18444(5)	305.22(13)	1.3447(6)	1.687(5)	1.563(5)
	pranlxd|1	0.18445(5)	305.23(13)	1.3448(6)	1.690(5)	1.562(5)
	CDAranlxd|1	0.18445(5)	305.23(13)	1.3448(6)	1.690(5)	1.562(5)
	pmtd	0.18446(5)	305.31(13)	1.3451(6)	1.685(5)	1.558(5)
	CDAmtd	0.18438(5)	305.10(13)	1.3442(6)	1.689(5)	1.563(5)
	CDAran64	0.18445(4)	305.28(13)	1.3450(6)	1.691(5)	1.564(5)
48	pranlxs|0	0.14943(5)	677.2(.4)	1.3261(7)	2.513(10)	2.313(8)
	CDAranlxs|0	0.14943(5)	677.2(.4)	1.3261(7)	2.513(10)	2.313(8)
	pmts	0.14931(5)	676.4(.4)	1.3246(7)	2.537(10)	2.333(8)
	CDAmts	0.14936(5)	676.6(.4)	1.3249(7)	2.521(10)	2.319(8)
	CDAran32	0.14930(5)	676.4(.4)	1.3244(7)	2.540(10)	2.335(8)
	pranlxd|1	0.14945(5)	677.4(.4)	1.3263(7)	2.522(10)	2.319(8)
	CDAranlxd|1	0.14945(5)	677.4(.4)	1.3263(7)	2.522(10)	2.319(8)
	pmtd	0.14946(5)	677.5(.4)	1.3266(7)	2.526(10)	2.319(8)
	CDAmtd	0.14941(5)	677.1(.4)	1.3259(7)	2.529(10)	2.326(8)
	CDAran64	0.14937(5)	676.8(.4)	1.3253(7)	2.510(10)	2.311(8)
64	pranlxs|0	0.12844(5)	1187.8(.7)	1.3083(8)	3.392(15)	3.109(13)
	CDAranlxs|0	0.12844(5)	1187.8(.7)	1.3083(8)	3.392(15)	3.109(13)
	pmts	0.12844(5)	1188.1(.7)	1.3086(8)	3.366(15)	3.089(13)
	CDAmts	0.12844(5)	1187.8(.7)	1.3083(8)	3.360(15)	3.094(13)
	CDAran32	0.12840(5)	1187.4(.7)	1.3078(8)	3.396(15)	3.116(13)
	pranlxd|1	0.12837(5)	1186.8(.7)	1.3072(8)	3.365(15)	3.088(13)
	CDAranlxd|1	0.12837(5)	1186.8(.7)	1.3072(8)	3.365(15)	3.088(13)
	pmtd	0.12855(5)	1189.5(.7)	1.3102(8)	3.383(15)	3.097(13)
	CDAmtd	0.12854(5)	1189.4(.7)	1.3100(8)	3.378(15)	3.100(13)
	CDAran64	0.12847(5)	1188.4(.7)	1.3090(8)	3.389(15)	3.113(13)

Table E.4.: Monte Carlo estimates of the absolute magnetization per spin and the magnetic susceptibility per spin from simulating the three-dimensional Ising model by means of the Metropolis algorithm on cubic lattices with extent L. We also considered different random number generators.

E.3. Execution times—Metropolis algorithm

The following table lists total per-replicum execution times of our simulations of the two-dimensional and the three-dimensional Ising model as well as the corresponding

mean update times per spin. Total execution times are given in seconds and are placed above the corresponding mean update times per spin, given in nanoseconds. As noted in Section 5.1.3, n-values larger than 20 may lead to little smaller update times per spin, compared to those depicted in Figures 5.5-5.6.

(L, n)	C1	C2	C3	C4	G1	G2	G3	G4	G5	G6
2D: $(16, 10)$	29.0 11.33	36.9 14.41	26.0 10.16	29.3 11.43	243.1 94.87	244.4 95.36	234.7 91.57	236.9 92.43	234.3 91.42	236.1 92.13
2D: $(32, 20)$	105 5.10	179 8.74	86 4.22	112 5.45	479 23.38	536 26.19	466 22.76	483 23.59	465 22.67	476 23.22
2D: $(64, 40)$	638 3.89	1172 7.15	483 2.95	675 4.12	877 5.35	1324 8.08	747 4.56	842 5.14	729 4.45	793 4.84
2D: $(128, 100)$	5833 3.56	11093 6.77	4359 2.66	6194 3.78	3195 1.95	7587 4.63	2097 1.28	3081 1.88	1917 1.17	2491 1.52
2D: $(256, 200)$	45091 3.44	87167 6.65	33294 2.54	47713 3.64	17827 1.36	52169 3.98	9700 0.74	15861 1.21	8258 0.63	11273 0.86
3D: $(16, 15)$	301 4.89	499 8.11	241 3.92	309 5.02	527 8.57	671 10.92	478 7.78	504 8.19	475 7.73	481 7.82
3D: $(24, 30)$	1709 4.12	3078 7.42	1357 3.27	1825 4.40	1311 3.16	2398 5.78	1050 2.53	1228 2.96	1008 2.43	1075 2.59
3D: $(32, 50)$	6637 4.05	11914 7.27	5162 3.15	7014 4.28	2950 1.80	7243 4.42	1917 1.17	2688 1.64	1737 1.06	2130 1.30
3D: $(48, 75)$	32850 3.96	58981 7.11	25384 3.06	34509 4.16	11863 1.43	33929 4.09	7134 0.86	10867 1.31	6139 0.74	8047 0.97
3D: $(64, 100)$	102771 3.92	185617 7.08	78388 2.99	107752 4.11	32509 1.24	100936 3.85	15992 0.61	27790 1.06	13109 0.50	18876 0.72

Table E.5.: Total per-replicum execution times of our simulations of the two-dimensional and the three-dimensional Ising model on lattices with periodic boundary conditions and extents L. Consecutive measurements were separated by n Monte Carlo sweeps. For each pair (L, n), a total of $1000000 = 10^6$ Monte Carlo estimates of system quantities were generated. We also considered different types of random number generators. For each entry the upper number refers to the total per-replicum execution time (in seconds), while the lower number refers to the mean update time per spin $\bar{t}$ (in nanoseconds).

E.4. On testing parallel random numbers

To obtain information about the quality of the parallel random number generators used in this thesis, we simulated the two-dimensional Ising model on a (16×16) (see Section 5.2.2) and a (32×32) squared lattice with periodic boundary conditions by means of the single-cluster algorithm with random numbers from different random number generators. For each generator we produced a total of 10 replica, each containing $100000000 = 10^8$ estimates of the internal energy and the specific heat. The statistical errors of these quantities were calculated with the `UWerr` script.

pranlxs|0	pranlxd|1	pmts	pmtd	CDAran32	CDAran64
1.433673(11)	1.433667(11)	1.433634(11)	1.433654(11)	1.433664(11)	1.433645(11)

Continued on the next page ...

	1.433671(11)	1.433671(11)	1.433666(11)	1.433668(11)	1.433671(11)	1.433655(11)
	1.433632(11)	1.433656(11)	1.433663(11)	1.433644(11)	1.433661(11)	1.433652(11)
	1.433655(11)	1.433661(11)	1.433656(11)	1.433662(11)	1.433655(11)	1.433672(11)
	1.433647(11)	1.433651(11)	1.433659(18)	1.433655(11)	1.433670(11)	1.433665(11)
	1.433656(11)	1.433647(11)	1.433648(11)	1.433659(11)	1.433657(11)	1.433648(11)
	1.433652(11)	1.433638(11)	1.433643(11)	1.433659(11)	1.433668(11)	1.433656(11)
	1.433663(11)	1.433659(11)	1.433666(11)	1.433649(11)	1.433668(11)	1.433647(11)
	1.433656(11)	1.433635(11)	1.433678(11)	1.433653(11)	1.433652(11)	1.433657(11)
	1.433644(11)	1.433676(11)	1.433665(11)	1.433649(11)	1.433661(11)	1.433669(11)
$-E$	1.4336548	1.4336562	1.4336577	1.4336552	1.4336626	1.4336568
error	0.0000035	0.0000035	0.0000036	0.0000035	0.0000035	0.0000035
dev.	-1.046σ	-0.646σ	-0.211σ	-0.931σ	1.183σ	-0.474σ
	1.8463(3)	1.8465(3)	1.8470(3)	1.8464(3)	1.8467(3)	1.8468(3)
	1.8464(3)	1.8468(3)	1.8471(3)	1.8466(3)	1.8467(3)	1.8470(3)
	1.8469(3)	1.8469(3)	1.8472(3)	1.8467(3)	1.8467(3)	1.8470(3)
	1.8474(3)	1.8467(3)	1.8465(3)	1.8469(3)	1.8470(3)	1.8464(3)
	1.8465(3)	1.8466(3)	1.8465(4)	1.8465(3)	1.8468(3)	1.8465(3)
	1.8466(3)	1.8473(3)	1.8472(3)	1.8465(3)	1.8471(3)	1.8466(3)
	1.8468(3)	1.8465(3)	1.8466(3)	1.8466(3)	1.8468(3)	1.8469(3)
	1.8469(3)	1.8469(3)	1.8471(3)	1.8470(3)	1.8467(3)	1.8468(3)
	1.8469(3)	1.8466(3)	1.8468(3)	1.8470(3)	1.8471(3)	1.8472(3)
	1.8468(3)	1.8464(3)	1.8466(3)	1.8472(3)	1.8469(3)	1.8462(3)
c_V	1.84676	1.84674	1.84686	1.84674	1.84685	1.84674
error	0.00009	0.00009	0.00009	0.00009	0.00009	0.00009
dev.	-0.084σ	-0.307σ	1.027σ	-0.307σ	0.916σ	-0.307σ

Table E.6.: Monte Carlo estimates of the internal energy per spin (the listed quantity is $-E$) and the specific heat per spin from Monte Carlo simulations of the two-dimensional Ising model on a (32×32) squared lattice using the single-cluster algorithm. Exact calculations [9]: $E = -1.43365846\ldots$ and $c_V = 1.84676759\ldots$

E.5. Critical exponents

The following tables list Monte Carlo estimates of the magnetic susceptibility χ, the Binder cumulant C_4, and $\partial Q/\partial\beta$ from simulations of the two-dimensional and the three-dimensional Ising model on squared/cubic lattices with periodic boundary conditions and extents L by means of the single-cluster algorithm. All quantities and their errors were calculated with the `UWerr` script.

L	C_4	χ	$\partial Q/\partial\beta$
32	0.61097(4)	207.15(3)	12.306(11)
40	0.61089(4)	306.09(4)	15.395(14)
48	0.61084(4)	421.19(6)	18.477(17)

Continued on the next page ...

56	0.61078(4)	551.52(8)	21.55(2)
64	0.61074(4)	696.70(9)	24.65(2)
80	0.61076(4)	1029.79(15)	30.76(3)
96	0.61072(4)	1416.7(2)	36.96(3)
128	0.61074(4)	2344.3(3)	49.23(5)
192	0.61070(4)	4765.8(7)	73.93(7)
256	0.61069(4)	7883.6(1.1)	98.53(10)
512	0.61066(4)	26516(3)	197.1(2)

Table E.7.: Monte Carlo estimates of the magnetic susceptibility χ, the Binder cumulant C_4 and $\partial Q/\partial\beta$ from simulations of the two-dimensional Ising model on squared lattices with extents L using the single-cluster algorithm.

	$\beta_c^* = 0.22165$			$\beta_c = 0.22165455$ (extrapolated)		
L	C_4	χ	$\partial Q/\partial\beta$	C_4	χ	$\partial Q/\partial\beta$
4	0.49484(4)	4.6987(5)	7.845(2)	0.49487(4)	4.6992(5)	7.845(2)
5	0.49005(4)	7.4826(9)	11.147(3)	0.49009(4)	7.4838(9)	11.147(3)
6	0.48650(4)	10.8735(13)	14.834(4)	0.48655(4)	10.8760(13)	14.835(4)
7	0.48376(4)	14.869(2)	18.899(6)	0.48383(4)	14.874(2)	18.901(6)
8	0.48180(4)	19.467(2)	23.295(7)	0.48188(4)	19.475(2)	23.298(7)
9	0.47999(5)	24.657(3)	28.058(9)	0.48010(5)	24.668(3)	28.062(9)
10	0.47869(9)	30.454(8)	33.10(2)	0.47881(9)	30.471(8)	33.11(2)
11	0.47753(9)	36.842(10)	38.44(3)	0.47767(9)	36.866(10)	38.45(3)
12	0.47660(10)	43.813(12)	44.13(3)	0.47676(10)	43.845(12)	44.14(3)
13	0.47576(10)	51.388(14)	50.04(3)	0.47595(10)	51.431(14)	50.05(3)
14	0.47495(10)	59.53(2)	56.34(4)	0.47517(10)	59.59(2)	56.36(4)
15	0.47449(10)	68.31(2)	62.65(4)	0.47473(10)	68.38(2)	62.68(4)
16	0.47387(10)	77.63(2)	69.40(5)	0.47413(11)	77.72(2)	69.43(5)
17	0.47342(11)	87.53(3)	76.36(6)	0.47371(11)	87.64(3)	76.40(6)
18	0.47292(11)	98.00(3)	83.60(6)	0.47324(11)	98.14(3)	83.65(6)
19	0.47256(11)	109.09(3)	91.07(7)	0.47290(11)	109.26(3)	91.13(7)
20	0.47206(11)	120.74(4)	98.74(7)	0.47243(11)	120.94(4)	98.80(7)
21	0.47184(11)	132.98(4)	106.56(8)	0.47225(11)	133.23(4)	106.64(8)
22	0.47147(11)	145.78(5)	114.72(9)	0.47191(11)	146.06(5)	114.82(9)
23	0.47130(11)	159.19(5)	122.98(9)	0.47177(11)	159.53(5)	123.08(10)
24	0.47091(11)	173.10(6)	131.66(10)	0.47141(11)	173.49(6)	131.79(10)
25	0.47081(12)	187.72(6)	140.31(11)	0.47134(12)	188.17(6)	140.45(11)
26	0.47043(12)	202.75(7)	149.31(12)	0.47100(12)	203.27(7)	149.47(12)
27	0.47032(12)	218.45(7)	158.55(13)	0.47093(12)	219.05(7)	158.73(13)
28	0.47007(12)	234.59(8)	167.89(13)	0.47071(12)	235.27(8)	168.09(14)

Continued on the next page ...

29	0.47007(12)	251.60(9)	177.42(14)	0.47075(12)	252.37(9)	177.65(15)
30	0.46973(12)	268.82(9)	187.2(0.2)	0.47044(12)	269.68(9)	187.5(0.2)
32	0.46926(12)	305.28(11)	207.4(0.2)	0.47006(12)	306.38(11)	207.7(0.2)
34	0.46914(12)	344.03(12)	228.1(0.2)	0.47002(12)	345.39(12)	228.5(0.2)
36	0.46863(12)	384.87(14)	249.7(0.2)	0.46960(12)	386.54(14)	250.2(0.2)
38	0.46835(12)	428.0(0.2)	272.2(0.2)	0.46940(12)	430.0(0.2)	272.7(0.3)
40	0.46804(13)	473.3(0.2)	295.4(0.3)	0.46918(13)	475.7(0.2)	296.0(0.3)
42	0.46781(13)	520.9(0.2)	318.6(0.3)	0.46904(13)	523.8(0.2)	319.4(0.4)
44	0.46760(13)	570.8(0.2)	343.3(0.3)	0.46892(13)	574.3(0.2)	344.2(0.4)
48	0.46696(13)	676.9(0.3)	393.8(0.3)	0.46849(13)	681.5(0.3)	395.0(0.6)
56	0.46633(13)	915.5(0.4)	502.3(0.5)	0.46828(14)	923.6(0.4)	504.2(1.1)
64	0.46541(14)	1187.8(0.5)	619.9(0.6)	0.46782(14)	1200.8(0.5)	623(2)
96	0.4628(2)	2615.7(1.3)	1171.2(1.2)	0.4674(2)	2670.4(1.3)	1182(15)
128	0.4597(2)	4551(3)	1839(2)	0.4670(2)	4702(3)	1867(65)

Table E.8.: Monte Carlo estimates of the magnetic susceptibility χ, the Binder cumulant C_4 and $\partial Q/\partial\beta$ from simulations of the three-dimensional Ising model on cubic lattices with extents L using the single-cluster algorithm. Estimates for $\beta_c^* = 0.22165$ result from Monte Carlo simulations, whereas values for $\beta_c = 0.22165455$ were extrapolated by means of the single histogram method, described in Appendix E.5.1.

fitting model: $\partial Q/\partial\beta = aL^{\frac{1}{\nu}}(1 + bL^{-\omega})$				
ω (fixed)	ν	a	b	χ^2/d.o.f.
0.75	0.62996(26)	0.8412(22)	0.100(6)	0.68
0.76	0.63002(26)	0.8418(22)	0.100(6)	0.68
0.77	0.63008(26)	0.8425(21)	0.099(6)	0.69
0.78	0.63013(25)	0.8431(21)	0.099(6)	0.69
0.79	0.63019(25)	0.8437(21)	0.099(6)	0.69
0.80	0.63024(25)	0.8443(20)	0.099(6)	0.70
0.81	0.63030(25)	0.8449(20)	0.099(6)	0.70
0.82	0.63035(25)	0.8454(20)	0.099(6)	0.70
0.83	0.63040(24)	0.8460(20)	0.098(6)	0.71
0.84	0.63045(24)	0.8465(19)	0.098(6)	0.71
0.85	0.63050(24)	0.8470(19)	0.098(6)	0.71

fitting model: $\chi = \chi_{\mathrm{a}} + cL^{2-\eta}(1 + dL^{-\omega})$					
ω (fixed)	η	χ_{a}	c	d	χ^2/d.o.f.
0.75	0.0365(5)	−0.224(6)	0.3439(7)	−0.165(7)	0.50
0.76	0.0363(5)	−0.222(6)	0.3436(7)	−0.166(7)	0.51
0.77	0.0362(5)	−0.221(6)	0.3434(7)	−0.168(7)	0.51
0.78	0.0361(5)	−0.220(6)	0.3431(7)	−0.169(7)	0.52
0.79	0.0359(4)	−0.218(7)	0.3429(7)	−0.171(7)	0.52
0.80	0.0358(4)	−0.217(7)	0.3426(7)	−0.172(7)	0.52
0.81	0.0357(4)	−0.215(7)	0.3424(7)	−0.174(7)	0.53
0.82	0.0356(4)	−0.214(7)	0.3422(7)	−0.176(7)	0.53
0.83	0.0355(4)	−0.212(7)	0.3420(7)	−0.178(7)	0.54
0.84	0.0354(4)	−0.211(7)	0.3417(6)	−0.180(8)	0.54
0.85	0.0353(4)	−0.209(7)	0.3415(6)	−0.182(8)	0.54

estimates of critical exponents deduced from scaling relations Eq. (B.14)				
ω	γ	α	β	δ
0.75	1.2369(6)	0.1101(8)	0.3265(7)	4.789(13)
0.76	1.2372(6)	0.1099(8)	0.3264(7)	4.790(13)
0.77	1.2374(6)	0.1098(8)	0.3264(7)	4.790(13)
0.78	1.2375(6)	0.1096(8)	0.3264(7)	4.791(13)
0.79	1.2378(6)	0.1094(8)	0.3264(7)	4.792(12)
0.80	1.2379(6)	0.1093(8)	0.3264(7)	4.793(12)
0.81	1.2381(6)	0.1091(8)	0.3264(7)	4.793(12)
0.82	1.2383(6)	0.1090(8)	0.3264(7)	4.794(12)
0.83	1.2384(5)	0.1088(7)	0.3264(6)	4.794(12)
0.84	1.2386(5)	0.1087(7)	0.3264(6)	4.795(12)
0.85	1.2387(5)	0.1085(7)	0.3264(6)	4.795(12)

Table E.9.: Least squares fits to the extrapolated values of χ and $\partial Q/\partial\beta$ for the three-dimensional Ising model. The lower sub-table gives estimates of the critical exponents γ, α, β and δ deduced from scaling relations.

E.5.1. The single histogram method

While for the two-dimensional Ising model the critical point is exactly known, for the three-dimensional Ising model only estimates of it are available. In Sections 5.1.3 and 5.2.3 we used $\beta_c^* = 0.22165$ from [2]. At present, there are more precise estimates of β_c [8]: $\beta_c = 0.22165455(3)$.

To extract system quantities at $\beta_c = 0.22165455$ from measurement data taken at $\beta_c^* = 0.22165$ (see Table E.8), the single histogram method was used [15]. The idea is to reweight estimates $\{X_{\mu_n}\}$ of quantity X, taken at inverse temperature β_0. The Monte Carlo estimate X_{MC} of X at inverse temperature β then is

$$X_{\mathrm{MC}} = \frac{\sum_{n=1}^{M} X_{\mu_n} \mathrm{e}^{-(\beta-\beta_0)E_{\mu_n}}}{\sum_{n=1}^{M} \mathrm{e}^{-(\beta-\beta_0)E_{\mu_n}}} \,. \tag{E.1}$$

One has to consider that the energies $\{E_{\mu_n}\}$ appearing in this equation are the total energies of the states $\{\mu_n\}$.

To get an idea of what is the largest difference $\Delta T = T - T_0$ that can be extrapolated over so that the method still gives reliable results, we consider the mean of the distribution $W(X(T))$ of the estimates of quantity X, which is $\langle X(T)\rangle$. It should fall within the range over which $N(X(T_0))$ is significantly greater than 1. Here $N(X(T_0))$ is the histogram of quantity X, sampled at temperature T_0. If we represent this range by the standard deviation σ_X of $N(X(T_0))$, then the criterion is

$$|\langle X(T)\rangle - \langle X(T_0)\rangle| \leq \sigma_X \,. \tag{E.2}$$

In the case of the internal energy, for instance, $\langle X(T)\rangle \approx E(T)$ and σ_E^2 is proportional to the specific heat. Thus,

$$|E(T) - E(T_0)|^2 \leq \sigma_E^2 = \frac{c_V(T_0)}{\beta_0^2} \,.$$

With the approximation

$$E(T) - E(T_0) \approx \left.\frac{\mathrm{d}E}{\mathrm{d}T}\right|_{T_0} \Delta T = c_V(T_0)\,\Delta T \,,$$

the distance that can reliably extrapolated away from T_0 is given by

$$\left[\frac{\Delta T}{T_0}\right]^2 = \frac{1}{c_V(T_0)} \,. \tag{E.3}$$

With respect to Table E.8, we assumed that similar conditions hold for χ and $\partial Q/\partial\beta$, so that extrapolating to $\beta_c = 0.22165455$ yields reliable results.

F. Simulating the Ising spin glass

In this chapter we introduce the parallel tempering method, commonly used to simulate (Ising) spin glasses on parallel computers. We also give a segment of our implementation of the checkerboard procedure for the two-dimensional bimodal bond distributed ISG.

F.1. The parallel tempering method

The idea behind the parallel tempering method is to perform several simulations of glassy systems in parallel, each of them with the same realization of the spin-spin interactions, but at different temperatures. From time to time one interchanges the spin configurations between two of the simulations with a certain probability which ensures that the states within each simulation still follow the correct Boltzmann distribution at the appropriate temperature. In this way, higher-temperature simulations (commonly at temperatures $T \gtrapprox T_g$) help lower-temperature simulations ($T < T_g$) to cross energy barriers, and so to evolve through their appropriate phase space.

Subsequently, we want all these simulations each to use the Metropolis algorithm to update their states, which in the case of the ISG works the same as for the Ising model. On the one hand, this enables us to draw on the methods from Section 5.1, especially with respect to perform the spin updates on parallel computer architectures, and on the other hand the use of the Metropolis algorithm ensures that the condition of detailed balance is obeyed for the Monte Carlo moves within each of the simulated ISG systems. To have the parallel tempering algorithm satisfy the condition of detailed balance, a possible choice of the acceptance probability for swapping the spin configurations between two systems, one at temperature T and the other at temperature $\tilde{T} > T$, is [15]

$$A = \begin{cases} \mathrm{e}^{-(\beta-\tilde{\beta})\Delta E} & \text{if } \Delta E = \tilde{E} - E > 0\,, \\ 1 & \text{otherwise.} \end{cases} \tag{F.1}$$

To prove that detailed balance is obeyed, we consider the joint probability

$$p_{\mu\nu} = \frac{1}{Z\tilde{Z}}\,\mathrm{e}^{-\beta E_\mu}\mathrm{e}^{-\tilde{\beta}E_\nu} \tag{F.2}$$

for the low-temperature system to be in state μ, while the high-temperature system is in state ν. As suggested by Eq. (F.2), we want $p_{\mu\nu}$ to reflect the desired Boltzmann distribution of the states in both systems. The condition of detailed balance then reads

$$\frac{P(\mu\nu \to \mu'\nu')}{P(\mu'\nu' \to \mu\nu)} = \frac{p_{\mu'\nu'}}{p_{\mu\nu}} = \frac{\mathrm{e}^{-\beta E_{\mu'}}\mathrm{e}^{-\tilde{\beta}E_{\nu'}}}{\mathrm{e}^{-\beta E_\mu}\mathrm{e}^{-\tilde{\beta}E_\nu}}\,. \tag{F.3}$$

Swapping the states between the two systems leads to $\mu' = \nu$ and $\nu' = \mu$. Thus,

$$\frac{P(\mu\nu \to \nu\mu)}{P(\nu\mu \to \mu\nu)} = \frac{\mathrm{e}^{-\beta E_\nu}\mathrm{e}^{-\tilde{\beta} E_\mu}}{\mathrm{e}^{-\beta E_\mu}\mathrm{e}^{-\tilde{\beta} E_\nu}} = \frac{\mathrm{e}^{-\beta \Delta E}}{\mathrm{e}^{-\tilde{\beta} \Delta E}}\,, \tag{F.4}$$

with $\Delta E = E_\nu - E_\mu$. According to Eq. (2.5), the transition rate $P(\mu\nu \to \nu\mu)$ can be split up into a product of the selection probability $g(\mu\nu \to \nu\mu)$ for the swap move, and the acceptance ratio $A(\mu\nu \to \nu\mu)$ for that move. If we assume these swap moves to be performed at regular intervals,[1] we can make $g(\mu\nu \to \nu\mu)$ a constant that finally cancels out. Equation (F.4) then becomes

$$\frac{P(\mu\nu \to \nu\mu)}{P(\nu\mu \to \mu\nu)} = \frac{A(\mu\nu \to \nu\mu)}{A(\nu\mu \to \mu\nu)} = \frac{\mathrm{e}^{-\beta \Delta E}}{\mathrm{e}^{-\tilde{\beta} \Delta E}}\,. \tag{F.5}$$

Obviously, this condition is obeyed by Eq. (F.1). Since the parallel tempering algorithm is designed to make low-temperature systems cross high-energy barriers and so to move from one energy basin to some other, it of course satisfies the condition of ergodicity.

A question that remains is about the number of updates that should separate the state-swapping. On the one hand, swapping states too often would make the involved low-temperature systems to be swapped back into the energy basins they recently escaped from. On the other hand, we want to swap states as often as possible since otherwise we would waste time doing the high-temperature simulations and not making use of their results. Since we want the high-temperature simulations to move away from the energy basins they are into after having performed a swap, the time-scale for the swapping moves is given by the energy autocorrelation time τ_E. The tempering algorithm should therefore attempt a swap move approximately every τ_E Monte Carlo sweeps, and the rest of the time perform normal moves.

F.2. Remarks on implementing the parallel tempering method

As detailed in the previous section, the parallel tempering method performs several simulations of glassy systems in parallel, each of them with the same realization of the spin-spin interactions, but at different temperatures. Since we want all these simulations use the Metropolis algorithm to update their spin configurations, implementing the parallel tempering method on a parallel computer is the same as implementing the checkerboard procedure for a large set of Ising spin glasses, except for the swap move. Since swapping two states just means to exchange pointers to the arrays the corresponding spin fields are stored in, we will focus on the Monte Carlo moves within any of the ISG systems. Except for the presence of ferromagnetic and anti-ferromagnetic spin-spin

[1] If the swap moves are performed each time after having updated the involved systems, say, M times in succession, the number of states, each of the two systems could have evolved to during the single-spin-flip updates, is the same for both of them. If there are N sites on the lattice, the probability for the system to move from some initial state to any other state is $1/(N^M)$, since none of the possible states is preferred to the others. Since this holds for both systems, g cancels out.

interactions, updating the spin configurations by means of the checkerboard procedure is almost the same as for the Ising model.

In the case of the bimodal bond distributed ISG, bonds can be created by mapping uniformly distributed random numbers onto the field $\{-1, +1\}$. Simulating the Gaussian bond distributed ISG requires to consider methods that provide normally distributed random numbers, which then can be assigned to the bonds. The Box-Muller transformation, for instance, can be used to transform two uniformly distributed random numbers, say, x_1 and x_2 into two normally distributed random numbers, say, y_1 and y_2:[2]

$$y_1 = \sqrt{-2\ln(x_1)}\,\cos(2\pi x_2)\,, \qquad y_2 = \sqrt{-2\ln(x_1)}\,\sin(2\pi x_2)\,. \tag{F.6}$$

Over the course of updating spin configurations, almost all computations involve these bonds. As a consequence, the number of memory requests increases. Since the graphics card's main memory is not cached, execution times of simulations using CUDA will strongly depend on the correct memory access patterns (see Appendix C.3) for the bond values. Execution times of simulations using the CPU should only slightly depend on memory access patterns—memory alignment is enforced by the compiler—.

The 2D bimodal bond distributed Ising spin glass—Implementations

As already mentioned in Chapter 6, we focused on the Metropolis update within any of the ISG systems, instead of implementing the parallel tempering method. Subsequent listing gives the source code of our CUDA kernel that implements the first update step of the checkerboard procedure (see Figure 5.1) for the two-dimensional bimodal bond distributed ISG. We incorporated performance issues from Appendix C.3.

```
    // lattice geometry
    #define LX 512
    #define LY 512
    #define NVOL (LX*LY)
 5  // thread block geometry
    #define SUB_X 64
    #define SUB_Y 2
    #define HALF_SUB_X (SUB_X/2)
    // fast integer multiplication?
10  #define MUL(x,y) __mul24(x,y)
    // double-precision?
    // DOUBLE
    #if defined DOUBLE
      #define FLOAT_TYPE double
15  #else
      #define FLOAT_TYPE float
    #endif

    static __global__ void
20  update_checkerboard_even(char *spin_field,
                             const char2 *bonds,
                             const FLOAT_TYPE *random_numbers)
```

[2] Since almost all random number generators produce uniformly distributed random numbers within $[0, 1)$, the logarithm in Eq. (F.6) will give an undefined value if evaluated for $x_{1,2} = 0$. The simplest way to overcome this issue is to skip all zeros within the random number stream. As a consequence, a small bias in the bond distribution may occur. In the following, we assume this bias to be negligible.

```
    {
     int
25    tx=threadIdx.x,
      ty=threadIdx.y,
      bx=blockIdx.x,
      by=blockIdx.y,
      g_id,
30    s_id,
      i_temp,
      sum;
     char2
      x_temp,
35
     __shared__ char sub_field[SUB_X*SUB_Y];
     __shared__ char sub_bonds[2*SUB_X*SUB_Y];

     g_id=by*SUB_X*SUB_Y*gridDim.x+ty*SUB_X*gridDim.x+bx*SUB_X+tx;
40   s_id=ty*SUB_X+tx;

     sub_field[s_id]=spin_field[g_id];
     sub_field[s_id+HALF_SUB_X]=spin_field[g_id+HALF_SUB_X];

45   x_temp=bonds[g_id];
     sub_bonds[2*s_id]=x_temp.x;
     sub_bonds[2*s_id+1]=x_temp.y;
     x_temp=bonds[g_id+HALF_SUB_X];
     sub_bonds[2*(s_id+HALF_SUB_X)]=x_temp.x;
50   sub_bonds[2*(s_id+HALF_SUB_X)+1]=x_temp.y;

     i_temp=(ty&1);
     read_id+=tx+i_temp;
     sub_id+=tx+i_temp;
55
     __syncthreads();

     // (+1)-direction
     if(i_temp==0){
60    sum=MUL(sub_field[s_id+1],sub_bonds[2*s_id]);
     }else{
      if(tx!=(HALF_SUB_X-1))
       sum=MUL(sub_field[s_id+1],sub_bonds[2*s_id]);
      else {
65     if(bx==(GRID_X-1))
        sum=MUL(spin_field[g_id-LX+1],sub_bonds[2*s_id]);
       else
        sum=MUL(spin_field[g_id+1],sub_bonds[2*s_id]);
      }
70   }
     // (+2)-direction
     if(by==(GRID_Y-1)){
      if(ty==(SUB_Y-1))
       sum+=MUL(spin_field[g_id-NVOL+LX],sub_bonds[2*s_id+1]);
75    else
       sum+=MUL(sub_field[s_id+SUB_X],sub_bonds[2*s_id+1]);
     }else{
      if(ty==(SUB_Y-1))
       sum+=MUL(spin_field[g_id+LX],sub_bonds[2*s_id+1]);
80    else
       sum+=MUL(sub_field[s_id+SUB_X],sub_bonds[2*s_id+1]);
     }
     // (-1)-direction
     if(i_temp==1){
85    sum+=MUL(sub_field[s_id-1],sub_bonds[2*(s_id-1)]);
     }else{
      if(tx!=0)
```

```
      sum+=MUL(sub_field[s_id-1],sub_bonds[2*(s_id-1)]);
     else {
90    if(bx==0)
       sum+=MUL(spin_field[g_id+LX-1],bonds[g_id+LX-1].x);
      else
       sum+=MUL(spin_field[g_id-1],bonds[g_id-1].x);
     }
95  }
    // (-2)-direction
    if(by==0){
     if(ty==0)
      sum+=MUL(spin_field[g_id+NVOL-LX],bonds[g_id+NVOL-LX].y);
100  else
      sum+=MUL(sub_field[s_id-SUB_X],sub_bonds[2*(s_id-SUB_X)+1]);
    }else{
     if(ty==0)
      sum+=MUL(spin_field[g_id-LX],bonds[g_id-LX].y);
105  else
      sum+=MUL(sub_field[s_id-SUB_X],sub_bonds[2*(s_id-SUB_X)+1]);
    }

    i_temp=MUL(sum,sub_field[s_id]);
110 if(i_temp<=0) {
     spin_field[g_id]=MUL(sub_field[s_id],-1);
    }else if(d_acceptance_ratios[i_temp]>random_numbers[g_id]){
     spin_field[g_id]=MUL(sub_field[s_id],-1);
    }
115 }
```

Listing F.1: CUDA kernel that realizes the first update step of the checkerboard procedure for the two-dimensional bimodal bond distributed Ising spin glass, as depicted in Figure 5.1.

For the most part the code is the same as for the two-dimensional Ising model (see Listing E.1 in Appendix E.1), except for the additional bond field `bonds`, which is copied into the shared memory (lines 45-50) the same way as the spin field `spin_field`. In lines 58-107, each CUDA thread then calculates the sum over its nearest-neighbor spins, where the bond values enter the arena. To speed these calculations up, our code provides the possibility to perform all integer multiplications by means of `__mul24(x,y)` (see line 10, which defines the macro `MUL(x,y)`) which performs much faster than the standard multiplication `x*y`. `__mul24(x,y)` calculates the product of the 24 least significant bits of the integer parameters `x` and `y` and delivers the 32 least significant bits of the result. The 8 most significant bits of `x` or `y` are ignored [17]. Lines 109-114 then implement the spin flip.

Bibliography

[1] V. Anselmi, G. Conti, and F. Di Renzo. GPU computing for 2-d spin systems: CUDA vs OpenGL. *PoS*, LATTICE2008:024, 2008.

[2] M. N. Barber, R. B. Pearson, D. Toussaint, and J. L. Richardson. Finite-size scaling in the three-dimensional Ising model. *Phys. Rev. B*, 32(3):1720–1730, Aug 1985. doi: 10.1103/PhysRevB.32.1720.

[3] P. R. Bevington and D. K. Robinson. *Data Reduction and Error Analysis for the Physical Science*. McGraw-Hill, 2002.

[4] J. J. Binney, N. J. Dowrick, A. J. Fisher, and M. E. J. Newman. *The Theory of Critical Phenomena—An Introduction to the Renormalization Group*. Oxford Science Publications, 1992.

[5] J. Cardy. *Scaling and Renormalization in Statistical Physics*. Cambridge University Press, 1996.

[6] P. D. Coddington and C. F. Baillie. Parallel cluster algorithms. *Nuclear Physics B - Proceedings Supplements*, 20:76 – 79, 1991. ISSN 0920-5632. doi: DOI:10.1016/0920-5632(91)90884-H.

[7] V. Demchik. Pseudo-random number generators for Monte Carlo simulations on Graphics Processing Units. `http://de.arxiv.org/abs/1003.1898v1`, 2010.

[8] Y. Deng and H. W. J. Blöte. Simultaneous analysis of several models in the three-dimensional Ising universality class. *Phys. Rev. E*, 68(3):036125, Sep 2003. doi: 10.1103/PhysRevE.68.036125.

[9] Ferdinand, Arthur, Fisher, and E. Michael. Bounded and Inhomogeneous Ising Models. I. Specific-Heat Anomaly of a Finite Lattice. *Phys. Rev.*, 185(2):832–846, Sep 1969. doi: 10.1103/PhysRev.185.832.

[10] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.*, 69(23):3382–3384, Dec 1992. doi: 10.1103/PhysRevLett.69.3382.

[11] M. Hasenbusch, A. Pelissetto, and E. Vicari. Critical behavior of three-dimensional Ising spin glass models. *Phys. Rev. B*, 78(21):214205, Dec 2008. doi: 10.1103/PhysRevB.78.214205.

[12] M. Lüscher. A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations. *Comput. Phys. Commun. 79 (1994) 100-110*, pages 0–19, 1993. doi: 10.1016/0010-4655(94)90232-1.

[13] M. Lüscher. User's guide for `ranlxs` and `ranlxd` v3.2, 2005.

[14] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/272991.272995.

[15] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, 1998.

[16] W. Nolting. *Grundkurs Theoretische Physik 6—Statistische Physik, 5. Auflage*. Springer Verlag, 2005.

[17] Nvidia Corp. Nvidia CUDA Compute Unified Device Architecture—Programming Guide, v2.0. `http://www.nvidia.com/object/cuda_develop.html`, June 2008.

[18] Nvidia Corp. Nvidia CUDA Compute Unified Device Architecture—Reference Manual, v2.0. `http://www.nvidia.com/object/cuda_develop.html`, June 2008.

[19] Nvidia Corp. `http://www.nvidia.com/object/cuda_showcase_html.html`, 2010.

[20] A. Pelissetto and E. Vicari. Critical Phenomena and Renormalization-Group Theory. *Phys.Rept.368:549-727,2002*, page 151, 2000.

[21] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *J. Comput. Phys.*, 228(12):4468–4477, 2009. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2009.03.018.

[22] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes—The Art of Scientific Computing*. Cambridge University Press, 2007.

[23] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On Testing GPU Memory for Hard and Soft Errors, 2009.

[24] M. Weigel. Simulating Spin Models On GPU; Talk given at ''35th Conference of the Middle European Cooperation in Statistical Physics Pont-à-Mousson'', March 18, 2010.

[25] U. Wolff. Comparison between cluster Monte Carlo algorithms in the Ising model. *Phys. Lett.*, B228:379, 1989. doi: 10.1016/0370-2693(89)91563-3.

[26] U. Wolff. Monte Carlo errors with less errors. *Comput. Phys. Commun.*, 156: 143–153, 2004. doi: 10.1016/S0010-4655(03)00467-3.

List of Figures

List of Tables

Acknowledgement

At first, I thank Prof. Dr. U. Wolff and Dr. H. Stüben for enabling me to work on this very interesting topic, for having been there for questions, and for having given suggestions for my work. I also want to thank PD Dr. M. Hasenbusch for answering questions on spin models. Also, I want to thank Ingmar Vierhaus for interesting discussions on physics and other topics, as well as for proofreading of this thesis. Sincere thanks are given to all remaining members of the Computational Physics group around Prof. Dr. U. Wolff for funny talks during the lunch and our group meetings.

At this point, I want to give thanks to Dr. T. Steinke and his workgroup at the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) for providing me with an account at the SGI Chippewa Falls GPU cluster, and also for helpful hints on GPU computing and hardware accelerators. Even, I want to thank the HPC workgroup around H. Busch and Dr. H. Stüben at the ZIB for providing me a student assistant position.

Special thanks go to my parents and my grandparents, who enabled me to finish my degree in Physics.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den 19.08.2010 Florian Wende